RL-TR-94-221
Final Technical Report
December 1994

ROME LABORATORY SOFTWARE ENGINEERING COOPERATIVE VIRTUAL MACHINE

Northeast Parallel Architectures Center

Geoffrey C. Fox and Salim Hariri (Syracuse University),
H.J. Siegel and H.G. Dietz (Purdue University),
and C.V. Ramamoorthy (University of California at Berkeley)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-221 has been reviewed and is approved for publication.

APPROVED:

CHESTER A. WRIGHT, JR., CAPT, USAF
Project Engineer

FOR THE COMMANDER:

HENRY J. BUSH
Acting Deputy Director
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Final ----

4. TITLE AND SUBTITLE
ROME LABORATORY SOFTWARE ENGINEERING COOPERATIVE VIRTUAL MACHINE

5. FUNDING NUMBERS
C - F30602-92-C-0150
PE - 62702F
PR - 5581
TA - 18
WU - P8

6. AUTHOR(S)
Geoffrey C. Fox and Salim Hariri (Syracuse University), H.J.Siegel and H.G. Dietz (Purdue University), (see reverse)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Parallel Architectures Center
111 College Place
Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-221

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Capt. Chester A. Wright, Jr./C3CB/(315) 330-4063

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This final technical report summarizes research accomplished by Syracuse University, Purdue University, and the University of California at Berkeley under the Expert Science and Engineering Program. The research accomplished a high-level analysis and feasibility study for building a virtual machine model for parallel software development. The report summarizes the three approaches (components, compiler techniques, and client-server) recommended by the researchers.

14. SUBJECT TERMS
Parallel processing, High performance computers, Parallel software development

15. NUMBER OF PAGES
76

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

6. (Cont'd)

and C.V. Ramamoorthy (University of California at Berkeley)

Final Report Contract No. F30602-92-C-0150

Rome Laboratory Software Engineering Cooperative Virtual Machine

Table of Contents

Vitrual Machine Project Summary	1 - 7
Syracuse University	8 - 43
Purdue University	44 - 53
University of California at Berkeley	54 - 67

Virtual Machine Project Summary

Geoffrey C. Fox

1 Introduction

The three subcontracts associated with the Virtual Machine address different aspects of the problem.

A. Syracuse - Overall System (engineering) issues for dividing problems into different components which can exploit data and task parallelism using scalable languages. Examples were implemented to illustrate this approach.

B. Purdue - A similar general area to Syracuse but with particular attention to necessary compilation techniques needed to efficiently address a variety of different target machines with both SIMD and MIMD architectures. In particular theoretical research and experiments based on the premise that the programs should be portable and the programming system should automatically minimize execution time by selecting and using the most appropriate target machine and execution mode (e.g., SIMD vs. MIMD).

C. Berkeley - This study focuses on a particularly critical aspect of the overall problem of expressing and then efficiently mapping heterogeneous problems onto heterogeneous target machines. This study presents an algorithm for mapping a dynamic network of processes onto a dynamic network of multiprocessors and workstations. The method is suitable for real time problems with the client-server paradigm.

2 Syracuse University Contribution

2.1 Introduction

A rather thorough study of the appropriate software models to portably represent complex problems is contained in Parallel Computing Works! by Fox, Messina and Williams, which will be published by Morgan Kaufmann in May 1994. In particular, Chapter 18 discusses "Metaproblems and Metasoftware" with an application to command and control. The previous chapters build up to this by discussing machine and software architectures for simpler

"component problems" which, together, make up the compound or metaproblems characteristic of the command and control applications. The Syracuse approach builds on the scalable languages such as High Performance Fortran (HPF) and similar C and C++ parallel extensions. Syracuse advocates coarse grain (object oriented) approaches to link the task parallel program modules together. Advanced Visualization System (AVS) has been successfully used for this in simple cases including those demonstrated on the ATM network between Rome Laboratory and NPAC which is a part of NYNET. The enclosed report discusses general issues and application to a simple problem.

2.2 Overview

The current advances in parallel and distributed software technology make the design of a Virtual Computing Environment attractive as well as cost effective. The ultimate goal of the Virtual Computing Environment is to utilize multiple heterogeneous computing platforms and dynamically allocate the resources needed to execute the user tasks. After analyzing the requirements of a wide range of applications, it can be shown that there is no single programming paradigm (e.g., data parallelism and/or function parallelism), programming language (e.g., Fortran, C, or C++ extension), computer architecture (e.g., SIMD, Shared-Memory/Distributed-Memory MIMD), and physical machine (e.g., TMC CM-5, Intel Paragon, or IBM SP-2) that can handle all the application requirements for computation, storage, communication, synchronization and visualization.

Syracuse's approach to build the Virtual Computing Environment has the following objectives:

1. Support an environment that utilizes multiple programming paradigms, programming languages, computer architectures, and physical machines instead of developing a new paradigm, a new language, a new computer architecture, or a new physical machine.

2. Promote software component reuse, i.e., the process of creating applications from existing software components rather than building them from scratch.

3. Provide a user-friendly human-machine interface that allows users to develop and execute the application, as wells visualize the result.

The virtual computing environment under development at Syracuse University consists of a number of layers, starting with the specification of the problem and its requirement and ending with an implemented solution of the problem. The intermediate levels of this system represent different levels of design and development, moving from a higher, architecture independent

level to a lower, machine specific level. At each layer, the developer is provided with tools that may be used to develop, test and evaluate the performance of the application from that layer's perspective. The approach is hierarchical and involves five layers: Problem Requirements Specification, Design Stage, High Level Virtual Machine (HLVM), Low Level Virtual Machine (LLVM) and Machine Level Implementation.

The Problem Requirements Specification layer is responsible for extracting the requirements of the problem to be solved and formalizing its functional flow. This layer will use a graph that identifies the major components of the problem and how these components are interconnected. The analysis performed by the Design Stage is based on Fox's work on the architecture of problems and portable parallel software systems. The goal of the Design Stage is to maintain a sufficient level of generality which enables the representation of a reasonable number of problems but retains specific information about the problem to allow its parallelization. The graph produced by the Design Stage feeds into the HLVM stage which is the first step in developing actual software. The software tools and languages to code the application at this level will be based on emerging standards (High Performance Fortran, High Performance C++, etc.). In the LLVM stage, the type of communications between the various nodes in the graph and the machine architectures for these nodes will be specified. The LLVM layer will support most of the major parallel/distributed communication tools that are currently available (e.g., PVM, Linda, Express, P4, MPI and ISIS). In the Machine Level Implementation layer, the solution of the problem is implemented for a specific set of machines. In this layer, the executables will be generated and executed. By identifying classes of machines that can run the various components of the program, the virtual computing environment will be able to provide the user with dynamic load balancing and fault tolerance capability.

3 Purdue University Contribution

3.1 Background

As parallel processing has developed, the execution models used with different systems have not converged. One is tempted to view this divergence as a sign of the field's immaturity; however, Purdue suggests that the use of a range of architectural designs, and hence of various execution models, is primarily driven by the fact that the relative performance of different system designs is highly application-dependent. Even individual functions within a parallel program often exhibit strong preferences for different system structures, and machines with the ideal structures may all be available within a single heterogeneous network. There is also the complication that, although a particular application might execute fastest when running

by itself on one system, the best turnaround time might result from running the program on a different system that is less heavily loaded at the time the job is submitted.

For these reasons, it is vital that programs be highly portable. However, portability usually sacrifices execution speed - programs are ported to new machines by simulating the intended target machine's features (e.g., a MIMD simulating a SIMD). Instead, Purdue proposes that portability and ease of program development be achieved by presenting the programmer with an abstract *programming model* from which programs can be mechanically transformed into effective code for any of a wide range of target machine *execution models*. Their report summarizes the progress that they have made toward these goals.

3.2 Overview

Purdue focused on the most basic issues that they identified in the original white paper. They concentrated on work that would help them define a plan for the research and development needed to make the virtual machine programming model a reality.

These issues center on the concept that a virtual machine programming model must be a complete, self-consistent, view of programming that is independent of the type of hardware used for execution. It even should be possible to debug one's program knowing only the programming model - without knowing the execution model used by the underlying super-computer (or network of computers). This can be accomplished easily by picking an execution model and having every other machine simulate that (e.g., MIMD executing SIMD code by inserting synchronization after each operation), but simulation sacrifices execution speed.

Thus, Purdue's research has been based on the premise that execution speed can be achieved by compiling programs directly into a native execution model for each target machine. Further, they want to use all the features of each target without making the features visible to the programmer, i.e., without making the program machine dependent and non-portable.

The wide variety of parallel execution models reflects application-dependent performance, and they want a single programming model to be able to target any execution model so that the best performance can be achieved. Because the handling of small-scale parallelism is relatively well understood, their initial work has emphasized the two most common types of large-scale parallelism:

- Multiple Instruction stream, Multiple Data stream (MIMD).
- Single Instruction stream, Multiple Data stream (SIMD).

In addition, Purdue does not want to require the user to decide which execution mode is

best. Instead, the system should automatically minimize execution time for each user program by

- Predicting performance for the program under each combination of execution model and target machine using both static (e.g., expected execution costs) and dynamic (e.g., load average) information.
- Picking the target(s) where the program has the shortest expected execution time.
- Causing the program to run there.

Although there is still much work to be done, the initial Rome funding has allowed Purdue to make significant progress toward this goal. A prototype AHS (Automatic Heterogeneous Supercomputing) has been implemented with the following features:

- AHS allows users to write their programs using a single programming model without any reference to or understanding of the target machine.
- Many execution models and target machines are supported.
- AHS performs flow-analysis based static cost estimation and compiles each user program into an executable script.
- When executed, the AHS-generated script considers dynamic loads and then automatically distributes, compiles, and runs the program on the target(s) expected to minimize execution time.

The programming model supported by AHS is an explicitly parallel dialect of C called MIMDC. This language uses a single-program multiple-data (SPMD) programming model supporting an arbitrary number of virtual processors. Processors communicate using shared memory references; synchronization across all processors is supported by a fast barrier synchronization construct. Although the MIMDC programming model is that of shared memory MIMD, the target systems currently supported by AHS range from a message-passing SIMD supercomputers to a network of UNIX workstations.

Purdue also conducted case studies to explore mapping algorithms to the different types of execution models that might be found in a heterogeneous suite of machines. Issues examined include trade-offs among SIMD, MIMD, and mixed-mode execution, performance prediction, the impact of changes in machine and problem size, the scalability of an approach, and the relationship between algorithm structure and the inter-processor communications network.

Considerable research work was also done toward extending AHS to break programs into segments that may be executed using different execution models and target machines, rather than treating entire programs as atomic entities.

4 Berkeley Contribution

4.1 Background

Berkeley presents an algorithm for allocating a dynamic network of processes on a dynamic network of multiprocessors and workstations communicating with a Client-Server model. Their effort has been to provide a real-time solution using only local information and computation. Although the problem of load sharing has been extensively researched, they believe that little work has been done in the context of this particular problem scenario. Among the important features of this algorithm is its adaptability, use of local computation only, and efficient distribution of load in real time. They expect this algorithm to be specifically useful in future Client-Server models in parallel systems that use multiple server threads to provide service.

4.2 Overview

This project develops a process allocation algorithm for the popular Client-Server model in a network of processors, where Server processors could be multiprocessors. The intended network is expected to be a cluster of multiprocessors and workstations connected by a high-speed network. The processor graph thus looks like a hierarchy of processor clusters. Such environments are already available for general purpose use and are expected to become widespread soon. Such an environment poses a variation to the existing problem of load sharing.

In existing models of Client-Server computation, the simplistic view of a service is one in which a process, or thread, is created for each client at the server site. However, with the evolution and widespread use of multiprocessors, it will be common to exploit parallelism and spawn multiple threads to do a single task. In other words, the simplistic view of the Client-Server model in the future will be a number of cooperating threads spawned in order to provide service for a single Client. In such a scenario, Berkeley expects that two kinds of problems will need to be solved - which server to request for a service (a macro-level load balancing), and how to spawn threads locally in order to provide a service effectively (a micro-level load balancing). This problem is a variant of the existing one, and they have not seen much literature on this problem. Their effort solves this problem, keeping two most important constraints in mind - Locality of Information and Computation, and Speed

in arriving at the thread allocation. Execution of a parallel computation on a network of processors involves two important phases - Task Allocation, and Task Scheduling, both well known NP hard problems. Task Allocation involves the mapping of processes onto processors. Task Scheduling involves scheduling the execution of processes on each processor. In the case of static networks, one can do the Task Allocation and Task Scheduling before runtime. However, for any application of even moderate size, static networks are clearly a very restrictive model. Their focus has been to provide constructive and useful algorithms which will be actually used in real-time applications. Thus, they consider dynamic networks of processes and processors. Consequently, Task Allocation and Scheduling must be done at run time.

Since their targeted applications are real time, they have two important aspects to consider - a good solution, and a fast solution. Most of the work in the literature has tried to find the best mapping algorithm, assuming a number of restrictions. Their focus is different. They want to see how efficiently they can map a process in real time, requiring as little information about the process or processor network as possible. Rather than trying to get as close to optimal as possible, introducing stringent restrictions (static network model, global knowledge of processor loads, process migration facility, etc.), Berkeley wants to see what they can do assuming as little as possible (dynamic network model, only local knowledge, no process migration during process execution, etc.). They call their algorithm the Dynamic Threshold Algorithm since the task allocation among processors is governed by a Dynamic Threshold at each processor.

Virtual Machine Project

Salim Hariri and Geoffrey Fox
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244-4100
hariri@cat.syr.edu, gcf@nova.syr.edu

Abstract

The current advances in parallel and distributed software technology make the design of a Virtual Computing Environment attractive as well as cost effective. The ultimate goal of the Virtual Computing Environment is to utilize multiple heterogeneous computing platforms and dynamically allocate the resources needed to execute the user tasks. After analyzing the requirement of a wide range of applications, it can be shown that there is no single programming paradigm (e.g., data parallelism and/or functional parallelism), programming language (e.g., Fortran, C, or C++ extension), computer architecture (e.g., SIMD, Shared-Memory/Distributed-Memory MIMD), and physical machine (e.g., TMC CM-5, Intel Paragon, or IBM SP-1) that can handle all the application requirements for computation, storage, communication, synchronization and visualization.

Our approach to build the Virtual Computing Environment has the following objectives:

1. support an environment that utilizes multiple programming paradigms, programming languages, computer architectures, and physical machines instead of developing a new paradigm, a new language, a new computer architecture, or a new physical machine.

2. promote software component reuse, i.e., the process of creating applications from existing software components rather than building them from scratch.

3. provide a user-friendly interface that allows users to develop and execute the application, as well as visualize the result.

The virtual computing environment under development at Syracuse University consists of a number of layers, starting with the specification of the problem and its requirement and ending with an implemented solution of the problem. The intermediate levels of this system represent different levels of design and development, moving from a higher, architecturally

independent level to a lower, machine specific level. At each layer the developer is provided with tools that may be used to develop, test and evaluate the performance of the application from that layer's perspective. Our approach is hierarchical and involves five layers: Problem Requirements Specification, Design Stage, High Level Virtual Machine (HLVM), Low Level Virtual Machine (LLVM) and Machine Level Implementation.

The Problem Requirements Specification layer is responsible for extracting the requirements of the problem to be solved and formalizing its functional flow. This layer will use a graph that identifies the major components of the problem and how these components are interconnected. The analysis performed by the Design Stage is based on Fox's work on the architecture of problems and portable parallel software systems. The goal of the design stage is to maintain a sufficient level of generality so as to enable the representation of a reasonable number of problems but retain specific information about the problem so as to allow its parallelization. The graph produced by the design stage feeds into the HLVM stage which is the first step in developing actual software. The software tools and languages to code the application at this level will be based on emerging standards (High Performance Fortran and High Performance C++). In the LLVM stage, the type of communications between the various nodes in the graph and the machine architectures for these nodes will be specified. The LLVM layer will support most of the major parallel/distributed communication tools that are currently available (e.g., PVM, Linda, Express, P4, and ISIS). In the Machine Level Implementation layer the solution of the problem is implemented for a specific set of machines. In this layer, the executables will be generated and executed. By identifying classes of machines that can run the various components of the program the virtual computing environment will be able to provide the user with dynamic load balancing and fault tolerance capability.

1 Introduction

Decades of experimentation with parallel/distributed computing has established its importance in handling real-world applications. Based on these premises, an enormous amount of research is being invested into exploring the nature of a general, cost-effective, scalable yet powerful computing model that will meet the computational and communication requirements of a wide range of applications that are encountered in C^3I applications as well as in Grand Challenge Problems (e.g., climate modeling, fluid turbulence, pollution dispersion, human genome, ocean circulation, quantum chromodynamics, semiconductor modeling, and superconductor modeling).

Active research in parallel processing has resulted in advances in all aspects of hardware

technology and software technology. Advances in hardware technology have resulted in complex, high speed processors, fast memories with large capacities, and high-bandwidth, low-latency interconnection network. Advances in software technology have provided easy-to-use tools and environments for the development of parallel applications. These advances have resulted in the proliferation of a large number of different architectural classes like SIMD computers, MIMD computers, vector computers, and data-flow computers, where each class represents a set of different trade-offs in design decisions like coarse-grain parallelism vs. fine-grain parallelism, shared-memory architecture vs. distributed-memory architecture, and circuit switched vs. packet switched. Each architectural class is tuned to deliver a maximum performance to a specific set of applications which it addresses. However it remains a fact that none of the existing computing systems are general enough to address all classes of applications and provide the desired performance levels. In addition, these architectures are not scalable and their relatively narrow applicability has prevented them from being cost-effective. Consequently, there is no single existing architecture that meets all of todays computing requirements. It is this realization that has spurred intense research in virtual computing environments.

We believe that the future of parallel computing lies in the integration of the plethora of specialized architectures into a single Virtual Computing Environment that allows them to cooperate in solving complex applications. The Virtual Computing Environment will capitalize on existing architectures and on current advances in computing technology to provide efficient, cost-effective, scalable, high-performance distributed computing. The objective of this project is to investigate the issues involved in the design of such an environment.

2 Trends Towards a Virtual Computing Environment

The main components of any heterogeneous distributed computing environment, that we refer to as a virtual computing environment, include a set of heterogeneous high performance computers, a programming environment, and a low-latency communication network. In what follows, we show how the current technological advances in these components will make the virtual computing environment meet the computing requirements of a wide range of complex and large real-world problems at a reasonable cost. Furthermore, we show that the diversity in problems architectures and their communications requirements supports the trend towards a virtual computing environment.

2.1 Trends in Computer Hardware

Trends in computer hardware can be captured by analyzing the advances in device technology and computer architecture.

2.1.1 Advances in Device Technology

Breakthroughs in semiconductor technology have allowed the development of denser and more sophisticated processors. For example, minimum feature sizes have dropped from ≈ 50 micron in the 1960's to 0.8 micron in the 1990's and is predicted to fall to 0.2 micron by the year 2000. Any decrease in the minimum feature size (f) is accompanied with an increase in the transistor density proportional to $\frac{1}{f^2}$ and an increase in speed proportional to $\frac{1}{f}$. Processor densities have increased from 1K in the early 1970's to over 1M in the 1990's. Basically, advances in processor design are heavily benefited from the development of IC technology, such as VLSI. VLSI technology allows the integration of massive processors with major reductions in board size, weight, and complexity. This has allowed multiple arithmetic and floating-point units, pipelines, memory management units, memory caches, DMA units and other functions to be integrated into the processor itself. Further, VLSI technology offers significant speed improvements over traditional equivalent processor implementation because VLSI packs more function into high speed clock zones and supports higher data/control communication bandwidth. Finally, VLSI technology reduces production cost by improving system-to-chip partitioning.

Another important underlying technology is advances in memory subsystems, including memory hierarchy, multiple memory modules, and management/access mechanisms. The objectives of these three issues are to enlarge the memory capacity and reduce the access time. For example, cache memories have been used mainly to bridge the speed gap between CPU and main memory. Multiple memory modules are designed to support multiple processors and thus provide the capability for parallel access. The management/access mechanisms (virtual memory and coherence protocols) address the basic issues of the memory subsystem. The direct impact of research on memory systems is shown by the existing shared-memory, distributed-memory, and distributed-shared-memory system organizations.

The third underlying technology is advances in interconnection networks, that offers high-bandwidth and low-latency communication for both processor-to-processor and processor-to-memory. The design of an interconnection network is based on a set of choices on network topology (static vs. dynamic), operation mode (synchronous vs. asynchronous), control strategy (centralized vs. distributed), and switching methodology (circuit switching vs. packet switching). In general, the design of an interconnection network is aimed at not only reducing

the latency and enhancing the throughput between the subsystems, but also at optimizing a set of parameters, such as maximizing the bisection bandwidth or minimizing the number of links. The existing systems (hypercube in nCUBE, fat tree in CM-5, or mesh in Paragon) have important improvements in the design of interconnection network.

2.1.2 Advances in Computer Architecture

The technology of the processor, memory, and interconnection network provides the major design space for developing the past, current and future high performance computer systems. According to [1], parallel computers today can be classified broadly as three basic approaches: *control-driven*, *data-driven*, and *demand-driven*.

- The *control-driven* (also referred to as von Neumann) approach consists of multiple von Neumann type uniprocessors and follows the traditional sequence-controlled cycle of fetch-execute-store using local and/or global memory. The computers of this class can be further classified by Flynn's taxonomy [2] into SISD, MISD, SIMD, and MIMD. SIMD architectures consists of vector/array processors, pipelined array processors, associative processors, orthogonal processors, and Multiple SIMD architecture. MIMD architectures consists of shared-memory MIMD, distributed-memory MIMD, systolic array architecture, and wavefront array architecture.

- The *data-driven* (also referred to as dataflow) approach is based on an execution paradigm in which instructions are enabled for execution as soon as all of their operands become available. Manchester Data Flow architecture, MIT Tagged Token Data Flow architecture, and Toulouse LAU System fall into this class.

- The *demand-driven* (also referred to as reduction) approach implements an execution paradigm in which an instruction is enabled for execution when its results are required as operands for another instruction already enabled for execution. University of North Carolina's FFP Machine is one example of the reduction machine.

However, each machine has different strengths/weaknesses and there is no single machine which is uniformly better than the others. Moreover, none of the above architectures is general enough to support all applications' requirements. This has driven computer manufacturers to develop new architectures which combine different types of parallelism into one machine. For example, PASM [3] was one of the earliest computers to incorporate heterogeneous modes of computation into a single architecture. The latest computer from Thinking Machines Corporation, the CM-5, combines the Data-parallel or SIMD model and the message-passing

or DM-MIMD model. Another example is the KSR 1 from Kendall Square Research which supports the SM-MIMD paradigm on top of a DM-MIMD architecture.

2.1.3 Discussion

In summary, the trends in device technology can be characterized in three features: 1) complex, high speed processing components; 2) fast memories with large capacities; and 3) high bandwidth and low latency interconnection network.

These advances have lead to an initial proliferation of a large number of powerful, high performance architectural designs with a somewhat narrow applicability. These systems, however, could not be used to solve many applications which combine several sub-problems with varied computational and communication requirements. For example, the global environment simulation problem requires vector supercomputers for fluid flow analysis, hypercube MIMD machines for contamination transport analysis, a scalar engine to model the temperature effects and workstations with graphic capabilities for real-time graphic visualization. This realization has lead to a gradual merging of architectural designs into single machines attempting to provide a broader applicability. However, this approach of building new architectures supporting multiple paradigms is simply not practical (in light of the current economy) considering the costs involved in designing and building new hardware, developing new software to support this design and finally training the user community to efficiently utilize the new design. Even if this was possible, it would be a slow process and would not be able to keep up with the increasing demands for higher computing power. The future of parallel processing lies in the integration of existing architectures and technologies into a powerful, scalable, general purpose computing environment which is efficient, cost-effective and capitalizes on current advances in computing and communication technologies. Such a heterogeneous network computing environment will be capable of delivering the required performance levels for general classes of applications. These trends towards the integration of existing architectures into a heterogeneous network computing environment are shown in Figure 1. The research presented in the following sections provides a methodology for achieving this goal.

2.2 Trends in Computer Software

In this subsection, we overview the techniques adopted for parallel/distributed computing. Also, we identify the software tools and programming environments that will enable the development of virtual computing environments. In order to provide a user-friendly and architecture-independent software programming environment, most research efforts focused on programming paradigms, programming languages, compiler technology, runtime systems,

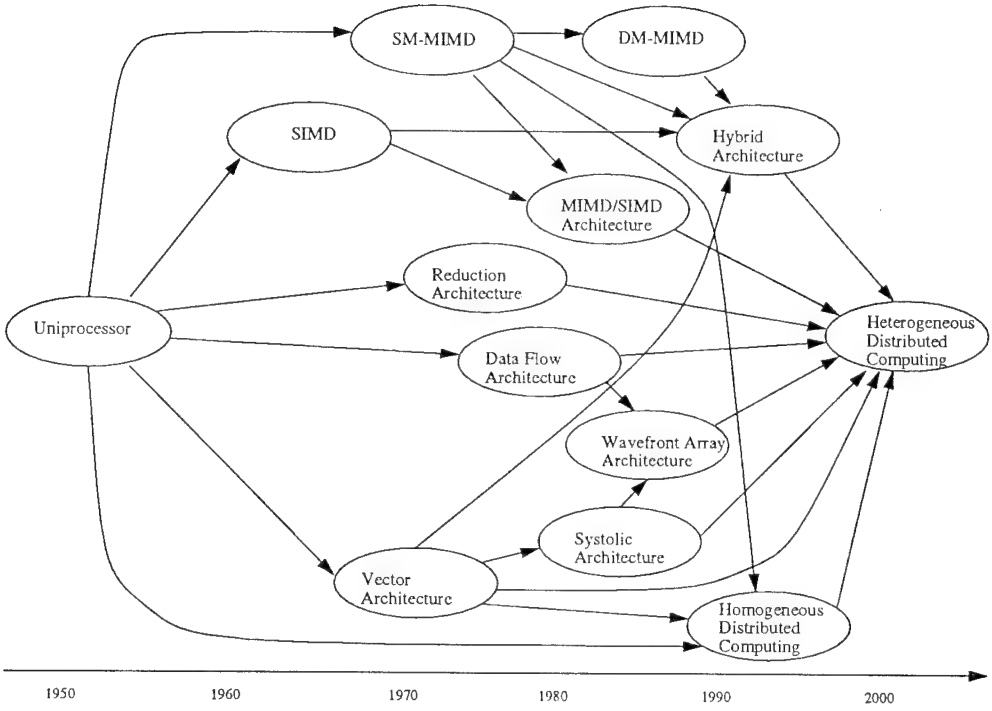

Figure 1: Architectural trends towards Heterogeneous Distributed Computing

and operating systems. Software tools can be broadly classified into three groups based on the service they provide to the programmer [4].

- The first group attempts to hide the parallelism from the user completely. These systems consist of parallelizing and vectorizing compilers which exploit the loop parallelism. The CFT compiler used in Cray X-MP, the KAP/205 compiler designed for Cyber-205, and Vienna Fortran Compilation System are three examples of this approach.

- The second group requires the user to explicitly write parallel programs by providing specialized extensions to existing languages. The language constructs could be based on the following: 1) data-parallel paradigm (vector and matrix operations); 2) shared-variable paradigm (semaphore and monitor); 3) message-passing paradigm (send, receive, and broadcast primitives); or 4) other supporting primitives (barrier, configuration, alignment, and distribution).

  This approach usually utilizes different technologies in library design, compiler development, and runtime support. Several examples based on different original programming languages are [5]

- Pascal-extension: Parallel Pascal.
- Fortran-extension: Fortran 90, Fortran D, HPF, CM Fortran, CMMD library, P4, PICL, Express, and PVM.
- C-extension: C*, DINO, CMMD library, P4, PICL, Express, and PVM.
- LISP-extension: *LISP.

- The third group provides a new programming language to implement parallel programs and its efforts can be characterized into three important trends [6].
  - In *Object-oriented programming language*, an object is used to integrate both data and the means of manipulating the data. The object-oriented concept becomes a powerful mechanism for parallelism because it can handle encapsulation and inheritance. Some existing programming languages in this class include CC++, Fortran-M, Mentat, Modular-2, and Pool2.
  - *Logic programming language* uses a formalized method of reasoning using inferences and deductions. There are several alternatives for introducing parallelism in logic programming languages that include *AND-parallelism, OR-parallelism,* and *parallel pattern matching.* Parlog and Strand are two examples that fall into this class.
  - *Functional programming language* is another approach for constructing parallel or distributed programs. Parallelism in functional languages is manifested through data dependencies and the semantics of primitive operators. SISAL, Id, and Val are examples that fall under this approach.

In what follows, we survey three main software fields that will play an important role in the development and the implementation of the virtual computing environments. These area include component-based software system, parallel/distributed programming environment, and network-based visualization system.

2.2.1 Component-based Software Systems

One of the important trends for building a large software system and exposing different characteristics within the system is based on the concept of software building blocks (also called components, modules, or objects). There are several advantages of a component-based software system. First, such systems can be easier to develop because components can be designed, programmed, and tested individually. Second, components can be stored using commercially available components for reusability. Consequently, the new application can be created by

reusing the existing components rather than building new ones from scratch. Third, components with well-defined interfaces can be written using different programming paradigms and programming languages and thus demonstrate one important aspect of application heterogeneity. Finally, such systems can run on parallel or distributed computing environments by assigning or replicating components to different processors.

There are several important issues that need to be addressed in a component-based system: *abstraction*, *selection*, *specialization*, and *integration* [7].

- *Abstraction* refers to an abstract model for classifying all the software components.
- *Selection* refers to a mechanism for locating a useful module from a collection of reusable components.
- *Specialization* refers to a procedure that adapts the selected module to fit the environment.
- *Integration* refers to a framework that combines a set of modules in the user's application.

Obviously, such methodology is very important in both the specification and configuration phase of our virtual machine conceptual model. The following is a brief description of several component-based software systems:

- *Module Interconnection Language* (MIL) [8] provides formal grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. Basically, a MIL can be considered a structural design language because it states what the system modules are and how they fit together to implement the system's function. MIL not only identifies the import and export of a module, but also performs static type-checking at an intermodule description level.
- *Advanced Visualization System* (AVS) [9] provides a graphical diagram for describing the interconnection of modules. Compared to MIL, which provides a formal machine-processable syntax to a user, the graphical diagram in AVS is more user-friendly. In AVS, modules are characterized by their input and output connections (source modules, transformation (filter and mapper) modules, and terminal modules). Selection and specification is done by separately searching the module palette and then changing the parameters through widgets.
- *Durra* [10] is a task-level description language designed to support the construction of distributed applications using concurrent, coarse-grained tasks running on networks of

heterogeneous processors. A Durra user describes an application as a set of components, a set of alternative configurations, and a set of conditional configuration transitions. The Durra's approach is focused on language, compiler, and runtime system. The runtime system is responsible for starting and terminating application processes, for passing messages between components, for monitoring reconfiguration conditions, and for carrying out reconfigurations.

- *Conic* [11] provides support for dynamic configuration for parallel and distributed programs. There are two languages in the Conic environment, one for programming individual task modules with explicitly defined interfaces, and one for the configuration of programs from groups of task modules. A graphics tool (ConicDraw) is used to display and manage the system. ConicDraw maintains a graphic representation of executing Conic systems in terms of the component instances which exist in the system, their interconnections and their execution state.

- *Draco* [12] is an approach for constructing software systems from reusable software components. In particular, Draco is focused on the reuse of analysis and design information in addition to programming language code. Several different mechanisms, such as source-to-source program transformations, module interconnection languages, software components, and domain-specific languages, work together to construct similar systems from reusable parts.

- *Agora* [13] is a software facility that supports the development of parallel applications written in multiple languages. At the core of Agora is a mechanism that allows concurrent computations on shared data structures independent of the underlying computer architecture. Agora shared memory can be implemented on both tightly and loosely coupled computer architectures.

2.2.2 Parallel and Distributed Programming Environments

- *Express* [14] is software programming environment for constructing parallel and distributed applications in an architecture independent manner. Express provides not only the configuration and communication library for C and Fortran, but also program debugging and performance analysis tools.

- *PVM* [15] is a software system that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. It provides C and Fortran libraries and is available on a large number of platforms. Heterogeneous Network

Computing Environment (HeNCE) [16] is an X-window based graphical interface tool built on top of PVM. It provides integrated tools for creating, compiling, executing and analyzing PVM programs.

- *Fortran-D* [17] (currently under development at the Northeast Parallel Architectures Center at Syracuse University and Rice University) is a set of machine independent compiler directives to Fortran77 and Fortran90. It is based on the concepts of "Annotated Complete Programs" where the programmer writes standard serial code and provides the compiler with data decomposition directives to achieve parallelism. This method provides an easy-to-use programming paradigm for the user while providing the flexibility and portability required to support varied and powerful architectures.

- *ISIS Distributed Programming Toolkit* [18] provides high level tools for developing reliable distributed applications.

- *Linda* [19] consists of a few simple operations designed to support and simplify the construction of explicit-parallel programs. Linda centers on an idiosyncratic memory model in which programs communicate through a logically shared associative memory called the tuple space.

- *Mentat* [20] is an object-oriented parallel processing tool based on C++. Mentat is designed to exploit the capabilities of both humans and compilers. The user is responsible for identifying object boundaries and specifying those object classes that have sufficient computational complexity to warrant parallel execution. The Mentat runtime system is responsible for managing all aspects of communication, synchronization and scheduling for the user.

- *SISAL* [6] is a general purpose functional language for parallel numeric computation. Constructions in SISAL are used to express scientific algorithms in a form close to their mathematical formulation with no explicit program control flow.

2.2.3 Network-based Visualization Systems

As we mention in section 2.4, visualization itself requires different software technology and hardware devices. As a result, combining existing visualization tools and powerful computation resources in a distributed system is one of the important trends in both academic and industrial communities. Below is a brief description of several ongoing projects.

- *An extension for AVS (Advanced Visualization System)* developed by [21] is to provide a heterogeneous remote procedure call facility, Schooner, that executes the computation task in a remote machine. To carry out the heterogeneity in different hardware platforms, Schooner provides three services: the Universal Type System (UTS), a collection of stub compilers, and a runtime system. In this environment, a module programmer has to hardcode the server name into the module specification as well as the end-user responses to select the site at runtime.

- The *ANL Visualization Project* [22] creates a system to implement common scientific visualization toolkits on the IBM SP-1 and SP-2 system. The target application is focused on the terabyte-sized datasets produced in Grand Challenge applications. Currently, AVS is used as a development tool for this project.

- *Sample Interface (SI)* [23] is used for viewing pixel images produced by a Cray supercomputer on a standard SUN workstation. There is no special hardware required in SI. The user interface on the Sun workstation is the standard Sun window system SunView. Data communication between the Cray and Sun is based on the client/server model and is implemented by BSD socket library over Ethernet. SI has the advantage of requiring only standard hardware and software.

- *The Scientific Visualization Workbench* [23] uses televisualization to move data from the supercomputer to the user and to modify the user's display. The workbench makes use of the Parallax board capability, in conjunction with supercomputer frame buffer system and the News window system. The communication between Cray supercomputer and Sun workstation is via Ethernet using TCP/IP protocol. Because of the special frame buffer system, the Scientific Visualization Workbench overcomes the speed limitations of other approach, but at an extra price.

- *Machine Graphics in Color (Magic)* [24] is designed to bring high-resolution, color images to the offices of researchers over a high-speed network. The Magic network transfers Cray graphics data up to one kilometer at a rate of 2.5 megabits per second. The display hardware of Magic is Tektronix 412x graphics terminals that provide high plotting speed, high resolution, a wide range of colors, picture segmentation, flexible graphics-input tools, fill-area, raster-plotting functions, and three-dimensional capabilities. The follow-on effort provides real-time animation and raster images on a Mac II.

- *The Advanced Visualization Research Project (AVRP)* [24] is a visualization environment that involves different research works on parallel graphics algorithms and visualization

system integration. The AVRP software system is supported on a wide variety of vendor hardware, such as Silicon Graphics, DEC, Sun, and Stellar platforms. One of the goals of this project is to integrate all the AVRP tools into a common environment through a language-based interpreter. On the other hand, all the AVRP efforts are based on unique and specialized LLNL-developed hardware.

- *Research on Interactive Visual Environments (Rivers)* [25] is developing hardware and software systems for extending high-end 3D visualization from a batch process to an interactive process, and for visualization-based interactive steering of supercomputing simulations in a high-performance distributed environment. The modular software subsystem of Rivers is executed across multiple, heterogeneous hosts. The communication network of Rivers is based on a 50-megabit-per-second HyperChannel network, an 80-megabit-per-second ProNet-80 network, and 100-megabit-per-second FDDI backbone network.

2.3 Trends in Communication Network

One basic requirement of distributed computing is high-speed interprocess communications over the underlying network. Fortunately, the transmission speed in communication networks increased by several orders of magnitude over the past decade as shown in Figure 2. One recent trend is focused on building networks in which the physical communication medium has a peak bandwidth on the order of 1 Gb/s or higher. The computer network architecture can be broadly decomposed into three components:

1. *the network component* that consists of the physical layer and medium access sublayer of the ISO/OSI Reference Model,

2. *the transport component* that consists of the data link, network and transport layers of the ISO/OSI Reference Model, and

3. *the application component* that consists of the session, presentation and application layers of the ISO/OSI Reference Model.

In what follows, we describe this trend and the suitability of the proposed new high speed transport protocols for the Virtual Computing Environment.

2.3.1 Network Component

Advance in computer networking technology, especially in fiber optic technology, has spurred intense research toward the design of a computer network capable of operating at speeds

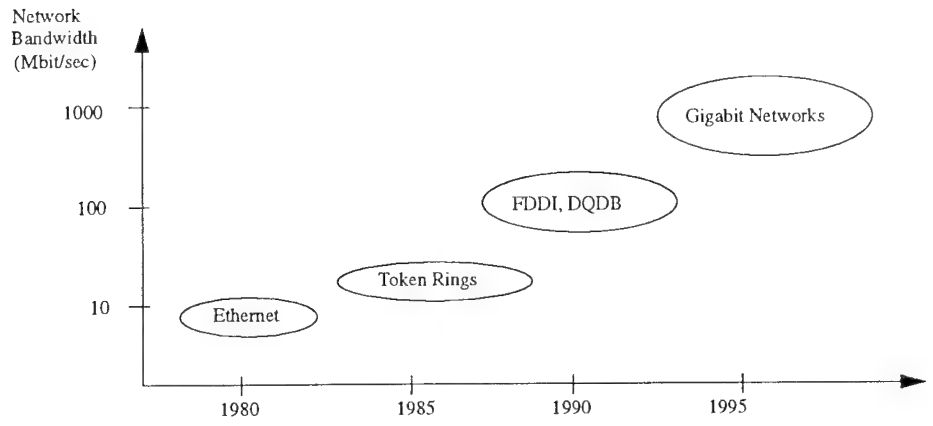

Figure 2: Trends in network technology

comparable to those offered by the transmission line. In what follows, we briefly highlight the features of a few high speed networks that have been recently designed and implemented.

- **FDDI**

  The Fiber Distributed Data Interface is a 100 Mbit/sec token-passing ring that uses optical fiber for transmission between stations and has dual counter-rotating rings to provide redundant data paths for reliability. Its potential applications include the interconnection of mainframes with their mass storage devices and other peripheral equipment as well as for backbone networks interconnecting lower speed LAN's.

- **DQDB**

  The IEEE 802.6 Distributed Queueing Dual Bus consists of two 150 Mbit/sec contraflowing unidirectional buses with slot generators at the head-ends that continuously send fixed-length time slots down the buses. Nodes access the time slots via a global distributed queueing algorithm.

- **HIPPI**

  HIPPI, the High-Performance Parallel Interface, is a copper-based data communications standard developed by ANSI X3T9.3 Task Group and capable of transferring data at 800 Mbit/sec (32 parallel lines) or 1.6 Gbit/sec(64 parallel lines). Almost all commercially available supercomputers and parallel machines support the HIPPI interface. In addition to the physical layer standard which applies to distances of 25 meters or less, a serial

HIPPI standard has been developed that will use optical fibers to extend HIPPI to distances as long as 10 km. In order to interface TCP/IP protocols, a set of data link layer standards is being developed as well. HIPPI is a point-to-point channel that does not support multidrop configurations.

- **SONET**

  The Synchronous Optical Network, also known internationally as Synchronous Digital Hierarchy (SDH), is a physical layer transmission standard. SONET is a series of optical signals that are multiples (called OC-N) of a basic signal rate of 51.84 Mbit/sec called OC-1 (Optical Carrier at level 1). The OC-3 (155.52 Mbit/sec) and OC-12 (622.08 Mbit/sec) have been designated as the customer access rates in future B-ISDN networks and signal rates up to OC-192 (9.953 Gbit/sec) are defined.

- **ATM**

  The Asynchronous Transfer Mode is the technique for transport, multiplexing, and switching that provides the high degree of flexibility required by B-ISDN. ATM is a layer 2 entity of OSIRM, on top of which one can add layers for building multilayer communication protocols. ATM is a connection-oriented protocol employing fixed-size packets (cells) with a 5-byte header and 48 bytes of information.

2.3.2 Transport Component

In spite of the proliferation of high-speed networks, the effective application bandwidth is still an order of magnitude lower than that provided by the network medium. Current transport protocols (TCP, TP4) were designed in 1970's with the assumption that the packet processing speed is faster than the packet transfer rates over the network; the transmission speeds were typically in the order of few Mbit/sec for LANs and 10-50 kbit/sec for WANs while the processing capacity was several MIPS. This assumption is no longer valid when we consider the current and future trends in network technologies. As network rates reach Gbit/sec range and higher, it will not be feasible for processors to process incoming packets at such high rates. This limitation has spurred intense research in the design of high speed transport protocols.

General design approaches for the high-speed protocols can be characterized as follows [26]:

1. new design approach: since current protocols were designed to be robust in the face of adverse network conditions, new protocols focus on simplifying the receiver process and streamlining the normal data transmission processing for maximum throughput;

2. architectural approach: this approach modifies the architecture of current implementation of the protocol layers. For example, the Xpress Transfer Protocol (XTP) combines layers 3 and 4 of the OSI reference model into a single software layer;

3. implementation approach: this approach implements either the standard protocols or new ones separate from the host by using special interface boards or interface processors.

In what follows, we summarize some of the recently developed transport protocols and high-speed network projects which are based on these three approaches.

- **NETBLT**

  The network block transfer protocol was developed to enable high throughput bulk data transfers to operate efficiently over long delay links such as those provided by satellites. NETBLT runs on top of the Internet protocol (IP) which provides an unreliable datagram service. The connections in NETBLT are unidirectional and normally can only be released by the sender. The unit of transmission is a buffer, several of which may be concurrently active to keep data flowing at a stable rate. Flow control and error control are performed separately and are different from conventional mechanisms based on window control with a transmitter-side timer (which is often hard to set due to the statistical distribution of round-trip delays, especially in wide area networks). Flow control is performed using rate control which limits the number of packet transmissions in a negotiated time interval. Error recovery is performed by selective retransmission with a receiver-side timer.

- **VMTP**

  The Versatile Message Transfer Protocol was developed to provide communication for the V distributed operating system over a network. VMTP aims primarily at supporting transaction-oriented communication which is based on request-response behavior. VMTP also offers a streaming mode which supports rate control by adjusting the inter-packet gap, and selective retransmission for efficient transmission of large amounts of data. Multicast and call forwarding functions are supported as well. Main features of VMTP are that network entities have location-independent identifiers and its use of address authentication for network security. The protocol is implemented on a Network Adapter Board (NAB) to offload the host processor.

- **XTP**

XTP was developed with the special design goal of VLSI implementation. It supports real-time datagrams and multicasting. The functions of OSI layer 3 and 4 are combined into one layer of XTP. XTP also provides a flexible addressing scheme, supporting the use of different address formats.

- **Protocol proposed by Netravali et al.** [27]

  Netravali *et al.* proposed a new transport protocol to support efficient communication for wide area networks. This approach employs the idea of periodic state exchange to simplify the protocol processing by removing some of the elaborate error recovery procedures. It also parallelizes protocol processing which runs on a separate interface board. Another feature of the protocol is a selective retransmission scheme based on packet blocking concepts which reduces the table management overhead usually required in selective repeat error-control methods.

- **Parallel implementation proposed by Schwartz et al.:** [28]

  ISO TP4 transport protocol is implemented on a multiple processor network interface board to parallelize the protocol processing. The main objective of the parallel implementation is to handle medium rates in excess of a Gbit/sec.

- **Nectar**

  Nectar is a high-speed fiber-optic network developed at Carnegie Mellon University and used as a network backplane to support distributed and heterogeneous computing. The Nectar system consists of a set of host computers connected in an arbitrary mesh via crossbar switches (HUB's). Each host uses a communication processor (CAB - Communication Accelerator Board) as its interface to the Nectar network. The network supports circuit switching, packet switching, and multicast communication through its 100 Mbit/sec optical links. Several transport protocols including the TCP/IP protocol suite and Nectar-specific transport protocols are implemented.

2.3.3 Discussion

Communication networks with high latencies and low application bandwidths have limited the applications that can efficiently run on network-based parallel/distributed computing environments in the last decade. The current advances in fiber technology have been able to stretch transmission rates from 8 Gbits/sec for 30 km distances in 1985 to over 10 Gbits/sec for 350 km distances in 1990 (a 16-fold improvement in 5 years). These advances have lead

Table 1: Advanced transport protocols and networks

Projects	Approach	Typical protocol suite	Medium Topology	Medium speed	Target Applications
NETBLT	New*	NETBLT/IP/802.5	Token Ring	10 Mb/sec	bulk data transfer
VMTP	New, HW*	VMTP/802.3	Ethernet	10 Mb/sec	rpc
XTP	Arch*, HW	XTP/FDDI	Token ring	100 Mb/sec	general
[Netravali]	New, HW	ATT*/IP/FDDI	Token ring	100 Mb/sec	general
[Schwartz]	HW	TP4/ IP/802.3	Ethernet	10 Mb/sec	general
Ultranet	HW	TP4/IP/HIPPI	point-to-point	1 Gb/sec	general
Nectar	New, HW	TCP/IP, RMP*/IP	point-to-point	100 Mb/sec	general
Safenet	Arch, HW	XTP/FDDI	token ring	100 Mb/sec	general

* New : new design
HW : hardware implementation
Arch : network architecture change
ATT : transport protocol by Netravali et al.
RMP : Nectar specific reliable message protocol

to aggressive research in the development of high speed computer networks and transport protocols. Table 1 summarizes the approaches used by recent projects to achieve these goals.

Furthermore, with the current movement towards the standardization of Gigabit LANs (ATM, Fiber Channel, and HIPPI), high-speed networks have received increasing industrial support. These trends will make high speed networks affordable and widely available. The advances in high-speed network technologies and their standardization will be a major driving force for virtual computing environments in the future.

2.4 Application Requirement

One important class of applications requiring a Virtual Computing Environment is the Grand Challenge problems and High Performance Computing and Communication (HPCC) applications. In general, these problems are characterized by massive data sets and complex operations that exceed the limits of current supercomputers. In order to further explore the requirements of these applications, we briefly examine the characteristics of such problems in terms of the following attributes: computation, storage, visualization, heterogeneity, parallelism, communication, and synchronization.

2.4.1 Computation Requirement

In general, solving the Grand Challenge problems is expected to require raw computational power between 100 and 1000 billions of operations per second. For example, [29] shows the

estimated computer requirements for several types of problems in turbulence physics research. To compute a flow over an entire aircraft using the complete Naiver-Stokes equations without approximation, a computer with a processing speed of 10^{19} operations per second is required to reduce the simulation time down to within 200 hours. Even in the approximation approach of flow analysis, such as large-eddy simulation, the expected computer speed is still in a teraflop per second range.

2.4.2 Storage Requirement

The Grand Challenge problems usually generate and/or might require to access a terabyte of data. For example, a mathematical model could produce large scale numerical data and graphical output for the purpose of simulation. However, an image understanding system could demand 2 or 3 dimensional, colored, high-resolution pixel data for lower-level processing. Consequently, two basic storage requirements for solving these applications are large-scale storage capacity and high-performance access mechanisms. Large-scale storage capacity in main memory and extended memory is used to keep pace with the I/O data size. High-performance access mechanisms (a low latency for scalar processing and a high bandwidth for vector processing) are used to keep pace with the CPU speed.

2.4.3 Visualization Requirement

Visualization of scientific data sets has become an important tool that allows scientists and engineers to efficiently and compressively understand the system behavior. In addition to the need of high-performance computing and storage, visualization itself requires special software technology and hardware devices. For example, most technology for displaying medical data fall into the three broad categories of surface-based rendering, binary voxel rendering, and volume rendering [30]. Each technology will involve different intermediate geometrical representations and different algorithms. Furthermore, specified hardware devices, such as stereo viewers, varifocal mirrors, cine sequences, real-time image-generation systems, and head-mounted displays are required for displaying the images. Obviously, it is impossible, or inefficient, to integrate all the software technology and hardware devices into a single physical machine.

2.4.4 Heterogeneity Requirement

The nature of the computations and communications characteristics of large complex applications is heterogeneous. For example, an image understanding system [31] [32] is usually classified into three levels (low, intermediate, and high) based on the data processed at each

level. Basically, image data, or corresponding arrays of numerical data, are processed in the lower level. Extracted image event, which is represented as symbolic description, is processed in the intermediate level. Finally, the high level will handle the knowledge data, such as semantic networks. Because the requirements at each level are very different in computation, communication, and control, each level demands different types of computer architecture, such as SIMD architecture for image processing in the lower level and MIMD architecture for semantic network traversal in the high level. Heterogeneity requirement has also been demonstrated in applications, such as computer robot, speech recognition, and C^3I (command, control, communication, and intelligence).

2.4.5 Parallelism Requirement

Parallelism has been exploited (from coarse-grain to fine-grain) into *five* different processing levels [33]: a) job or program level; b) subprogram or job step level; c) procedure, subroutine, task, or coroutine level; d) non-recursive loop or unfolded iteration level; and e) instruction or statement level.

Each parallelism level (or granularity) has different characteristics and requires different approaches to exploit parallelism at that level. For example, fine-grain parallelism (levels (d) and (e)) usually requires a parallelizing compiler and SIMD computer. On the other hand, coarse-grain parallelism (levels (a) and (b)) usually requires effective operating system and MIMD computer. However, a combination of these levels may be required to achieve the best performance. Consequently, a general-purpose parallel computer should be able to support all the features used to exploit parallelism that range from fine-grain to coarse-grain.

2.4.6 Communication Requirement

Communication patterns in different applications (*regular* or *irregular*) not only provide a strong impact on selecting the best computer architecture, but also affect the performance of partitioning, mapping, and scheduling tasks. Regular communication patterns, such as butterfly operation in FFT, nearest neighbor communication in Laplace's equation, or all-pairs communication in N-body problem, can be easily satisfied by mesh, ring, hypercube or other such architectural topologies. On the other hand, irregular communication patterns (sparse matrix computation, combinatorial searching, and discrete event simulation) are more difficult to map to existing architectures.

2.4.7 Synchronization Requirement

Fox [34] presents a classification for problem architecture which can be used to identify appropriate hardware and software paradigms. Fox's classification is based on a break up of each problem into *spatial* (data) and *temporal* (control) aspects. The temporal structure of a problem is analogous to the hardware classification into SIMD and MIMD. The spatial structure of a problem is analogous to the topology of the hardware. The problem structures can be classified into five categories: 1) synchronous, 2) loosely synchronous, 3) asynchronous, 4) embarrassingly parallel, and 5) loosely synchronous complex.

Each problem structure requires different hardware and software to solve it. For example, synchronous problems are inherently data parallel and therefore it maps naturally into a SIMD architecture. Asynchronous problems are inherently functional parallel and thus they map naturally into a MIMD architecture.

3 History of Virtual Machine Concept

In order to develop the Virtual Machine concept on a heterogeneous computer network, we will examine the original idea of IBM's Virtual Machine project (VM/IBM). The most important parallel between IBM and our approach is that they are both novel implementations of software structures intended to present the user with the most useful programming paradigm that can be supported on the given hardware. The following discussion will illustrate this statement more clearly.

The current version of IBM's Virtual Machine operating system is called VM/ESA (Virtual Machine/Enterprise Systems Architecture). It is the latest in a series of operating systems intended to increase the utility of large IBM mainframe computer systems. An understanding of the evolution of operating systems is helpful in explaining the motivation for current efforts to design new paradigms for configuring, programming and running modern computer systems.

When computers were first developed in the 1950's, a user would write a program, reserve time on a machine, run the program (without the benefit of modern conveniences like operating systems, linkers or compilers), gather up the results and leave the machine to the next person. This resulted in a 1 to 1 relationship between computers and programs. This paradigm was inefficient. The computer spent too much of its time idle between users.

In the 1960's multiprogramming operating systems were developed (like IBM's OS/360). They supported a new paradigm. Now several user programs could reside in memory at the same time. The operating system allocated system resources among the various jobs to achieve very high hardware utilization. In this new paradigm the relationship between computers and

programs was 1 to many.

While IBM's first multiprogramming operating systems (which evolved into MVS) made life easier for applications programmers, the people developing operating systems (system programmers) still had a pretty tough life. A programmer writing a new operating system module would still have to reserve an entire computer to bring up his own system and test his module. This inefficiency was due to the fact that the relationship of computers to operating systems was 1 to 1. This led to the development of IBM's Virtual Machine technology. VM/IBM is a mainframe hypervisor which presents (through a combination of simulation and emulation) each user with the image of an entire machine. This allows every user to boot their own operating system. Besides being helpful to systems programmers, VM also provides other advantages. An installation that desires to run applications that require different mainframe environments can set up several environments in different virtual machines running on a single mainframe. Also, VM provides a (relatively) user friendly interface through a single user operating system (VM/CMS) that runs in a virtual machine.

While VM/IBM provides a flexible mainframe programming paradigm, the computing environment of choice is no longer centered on mainframe technologies. Due to factors like high cost, centralized administration, fault intolerance, and poor usability, mainframe systems are rapidly losing their markets to smaller, workstation based computing platforms. The evolution of operating systems for small machines is similar to that of large machines. This evolution cumulated with UNIX, a multiprogramming system for small computers. There is nothing comparable to VM/IBM for workstations because there is no need to provide multiple system images on a small computer. That is, on small systems a 1 to 1 relationship between computers and operating systems is adequate.

The current computing environment is drastically different from what was predominant 15 years ago. A single mainframe, or a few stand-alone mainframes, could support the vast majority of an entire organization's computing. Now, most environments include a vast array of hardware platforms. Everything from portable single user systems to scientific supercomputers are employed. Commonly, wide varieties of systems are interconnected. This new, heterogeneous paradigm faces many challenges never encountered in the mainframe world. Two of particular interest are

- Making optimal use of an organization's resources. A great deal of computing power is wasted in underutilized single user systems.
- Maximizing program performance by executing each task on the most appropriate hardware platform.

The solution to these problems may lie in a new software paradigm. One that supports a many to many relationship between computers and programs. A system that provides an intuitive mechanism for users to specify the tasks and to transparently distribute them across the available computing resources.

4 A Proposed Virtual Computing Environment

The main design philosophy of our approach for creating a Virtual Computing Environment is to utilize multiple existing programming paradigms, programming languages, computer architectures, and physical machines, instead of developing a new paradigm, a new language, a new computer architecture, or a new physical machine, that provide high performance computing for a wide range of applications. In order to achieve this goal, we describe the software development process for our Virtual Computing Environment. Figure 3 highlights the main stages of the software development process for our Virtual Computing Environment. Conceptually, the software development process can be viewed as a transition of stages, starting with the specification of the problem (Problem Specification) and ending with an implemented solution of the problem on a set of heterogeneous computers (Machine Level Implementation). The intermediate stages of this development process represent different levels of design and development, moving from a higher, architecture-independent level to a lower, machine-specific level. In addition to showing the stages of the software development process, Figure 3 shows, for each stage, the associated issues/features, the current status and emerging software support.

The Problem Specification stage of the software development process is responsible for formalizing the structure of the problem. The Design Stage analyzes the problem characteristics and its architecture. The subsequent two stages (High Level Virtual Machine and Low Level Virtual Machine) represent a transformation of the architecture-independent, abstract solution into a more detailed, architecture-specific solution of the problem, as we go down towards the Machine Level Implementation stage. The final stage (Machine Level Implementation) uses machine specific software to implement the problem solution and incorporates details specific to the target system (CM5 and Paragon) and programming language (Fortran plus message passing and Ada). In what follows, we describe the stages of the software development process.

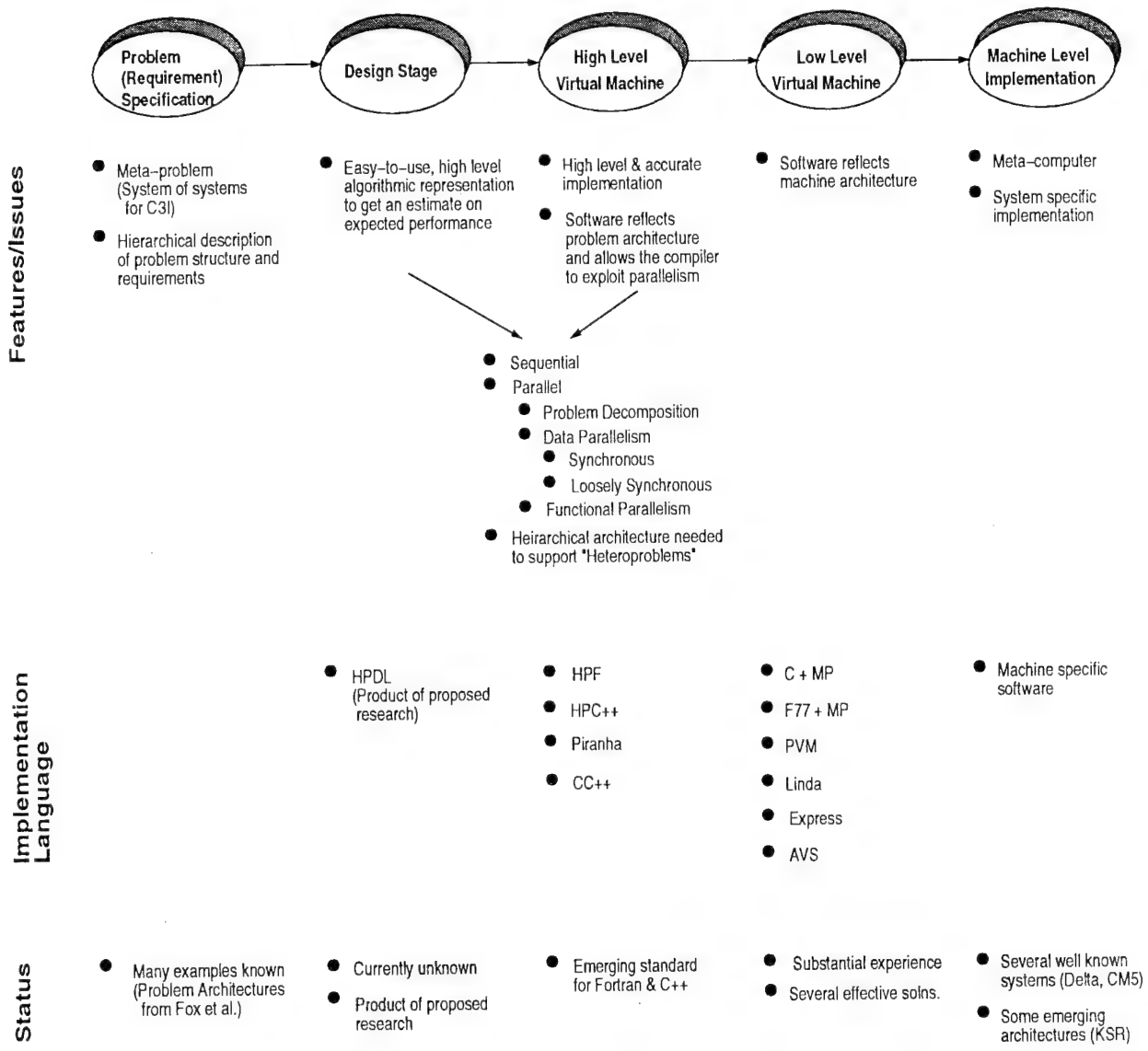

Figure 3: Software Design Process for Virtual Computing Environment

4.1 Problem Specification Stage

The problems to be addressed in this project belong to the class of large, complex applications, such as Grand Challenge problems or C^3I systems. Such applications are called *meta-problems* and can be structured hierarchically as a system of subproblems. Therefore, the Problem Specification stage of the software development process will use a specification tool to decompose the application into a set of software building blocks, called *tasks*, and formalize its functional flow. Basically, the Problem Specification stage exploits the functional parallelism at the top level of the application. The inherent functional or data parallelism

within each task will be examined in the next stage.

The suitable software tool of the Problem Specification stage can be classified into two classes, i.e., textual approach and graphical approach. Textual approaches, such as MIL [8], provide formal grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. Graphical approaches, such as AVS's Network Editor [9], allow a user to utilize icon constructs for describing the application.

4.2 Design Stage

The Design stage is responsible for analyzing the problem specification, exposing the inherent parallelism for each task, and utilizing a High Performance Design Language (HPDL) to express data parallelism and functional parallelism. In this stage, each task is basically a functional component of the application, demonstrates different characteristics (sequential, data parallelism, functional parallelism, or visualization), and requires a certain combination of paradigms, languages, architectures, and machines to achieve the best performance. However, the major goal of HPDL is to assist the user in experimenting with and evaluating the different parallel design alternatives, such as decomposition approach, task granularity, and parallelism type (synchronous/loosely synchronous data parallelism, functional parallelism, or a combination of both), based on the task characteristics. Our analysis of the task's characteristics is based on Fox's work in problem architectures and portable parallel software systems [34]. First, we determine whether or not a task can be parallelized. If the task can be parallelized, we consider a break up of this task into *temporal* (control) and *spatial* (data) aspects for investigating the type of parallelism. The temporal structure is analogous to classifying hardware into SIMD and MIMD. Further detail is contained in the spatial structure describing the problem at a given instance of its execution time. The spatial structure is analogous to the interconnect or topology of the hardware. Based on the temporal and spatial structure, the characteristics of task can be divided into five classes, i.e., synchronous, loosely synchronous, asynchronous, embarrassingly parallel, and loosely synchronous complex, that demonstrate functional parallelism or data parallelism.

HPDL, which is a specification language and not a compilable language, will provide the user with the capability to express the temporal and spatial characteristics of the problem architectures. Furthermore, HPDL simplifies the process of quantifying and evaluating different parallel designs of the application. The HPDL specification would encapsulate the features of two types of design languages, Data Parallel Design Language (DPDL) and Functional Parallel Design Language (FPDL). DPDL is mainly for the specification of data parallelism available

in different tasks of the application. The syntax of the parallel constructs in DPDL needs the functionality of ALIGN, DISTRIBUTE, TEMPLATE, FORALL, DO INDEPENDENT, and other HPF commands. FPDL is used to express the functional parallelism available in the tasks. The syntax of the parallel constructs is based on HPC++, which uses C++ as a basis for expressing functional parallelism.

4.3 High Level Virtual Machine

In High Level Virtual Machine, a task is implemented by an architecture-independent programming language that corresponds best to the problem characteristics, its architecture, and the suitable programming paradigm described by HPDL. The software tools and languages to code and parallelize the application at this level will be based on emerging standards (High Performance Fortran HPF, and High Performance C++).

In this stage, we allow different implementations for a given task to coexist in the environment. There are several reasons why multiple implementations should be provided to a given task. First, the best algorithm for a given task usually depends on the problem size and the number of processors. For example, the performance of a distributed Dijkstra algorithm for all-pairs shortest path problem is better than distributed Floyd algorithm if the number of vertices is small and the number of processors is large. On the other hand, if the number of vertices is large and the number of processors is small, the performance of a distributed Floyd algorithm overwhelms distributed Dijkstra algorithms [35]. The scalability issue is a common problem in parallel/distributed processing. Our approach for solving this problem is not to develop a new algorithm suitable for any problem size and any number of processors, but provide these two algorithms to the environment simultaneously and let the system select the best one at runtime. In addition to the problem size and the number of processors, the characteristics of input data are another factor that effects the selection of algorithms. For example, different algorithms for matrix operations could depend on the characteristics of the input matrix, such as sparse, banded, or dense [36]. Again multiple algorithms should be utilized in order to capitalize on the characteristics of input data. Finally, from the perspective of utilization and load balancing in network resources, multiple implementations for a given task provide the opportunity to choose the best implementation at runtime.

4.4 Low Level Virtual Machine

In this stage, the architecture-independent programs developed in High Level Virtual Machine will be transferred into an architecture-specific style. In other words, the implementation of a

task on the existing programming tools reflects the required programming paradigms and the underlying computer architectures without any machine detail. For example, a task could be implemented by using message passing programming paradigm (C or Fortran plus message passing primitives or C++) to represent the task functional parallelism and can be executed on a distributed-memory MIMD machine or a cluster of workstations.

Because some of the existing software programming tools provide high portability across different hardware platforms, the mapping mechanism between Low Level Virtual Machine and Machine Level Implementation will be more flexible and straight forward. For example, Express [14] provides portable message passing library for standard C or Fortran, and can run on a Cray supercomputer or a cluster of IBM RS/6000 workstations. As a result, the same Express source code could produce two object codes, one for Cray and one for IBM RISC/6000 workstation. It is also important to note that the same object code could be running on different physical machines.

4.5 Machine Level Implementation

In this layer, programs in Low Level Virtual Machine will be mapped into the Machine Level Implementation by compiler technology and runtime mechanism. Complier technology will transfer high-level message passing primitives into low-level machine-specific message passing primitives. Further, compiler technology also analyzes data and/or control dependency to vectorize and parallelize the source code into machine-specific statements or instructions. The runtime mechanism will handle data decomposition and distribution, process-to-processor mapping and scheduling, and exception handling.

5 A Case Study: Colloidal Dispersions

In this section, a scientific application is presented to demonstrate the idea of the Virtual Computing Environment. The goal of the application is to study the stability of dispersions in a colloidal system by Monte Carlo simulation. This application is chosen for the following reasons:

- The main body of this application (Monte Carlo simulation) is a computationally intensive problem.
- This application can be decomposed into several components, and each component requires different programming paradigms to achieve its best performance.

- The numerical result of the application must be visualized in order to better understand the system behavior.

In what follows, we describe this application and how it can be implemented using the resources available in the proposed Virtual Computing Environment.

Research in colloidal science is very important in various branches of pure chemistry, biology, medicine, agriculture, industry, and many other fields. Colloids are substances consisting of a homogeneous medium and of particles dispersed therein. Indian ink, a soapy shaving cream, blood serum, and glue are all examples of colloids. One basic characteristic of colloid is that colloidal particles are smaller than coarse, filterable particles but larger than atoms and small molecules. In this study, we are interested in the spatial arrangement or configuration of the particles, described by Radial Distribution Function (RDF), in a colloid. Usually, in the whole process of analysis, RDF is an intermediate result and some thermodynamic properties can be calculated in terms of RDF.

Practically, a radial distribution function, $g(r)$, measures the particle density as a function of distance r from an arbitrary central particle. One of the theoretical approaches to calculate $g(r)$ is Monte Carlo method. In general, Monte Carlo method refers to a process that employs pseudo-random number generators to simulate physical systems which are inherently probabilistic or statistical in nature. The scenario of Monte Carlo method for calculating $g(r)$ is briefly given as follows. We start from some initial spatial configuration and then a subsequence of configurations are generated by moving the particles one at a time. As time goes on the spatial configuration changes continuously and simulates the effect of thermal agitation. In our analysis, three different initial spatial configurations (randomly-generated, face-centered, and previous-running) are taken into account in order to compare their effects and the final results. The movement of particles are based on three interparticle forces (van der Walls forces, electrostatic double layer forces, and steric forces). The radial distribution function is calculated by averaging all the chosen configurations, known as ensemble average, and is visualized by plotting the graph of $g(r)$ against r.

This application has been implemented sequentially and in parallel. The sequential version is implemented in C and is running on a VAX/VMS. The parallel version is implemented in C and CMMD library and is running on the CM-5. It is important to note that the visualization of RDF is off-line in the previous implementations. In other words, the numerical data of RDF is stored in a file and then a graphical tool is utilized to plot the numerical data. The heterogeneous, distributed version based on the model of Virtual Computing Environment for this application will be described in the next section.

5.1 The Structure of Application

Figure 4 shows the application structure of colloidal dispersions. Figure 5 shows the application structure represented by AVS's Network Editor.

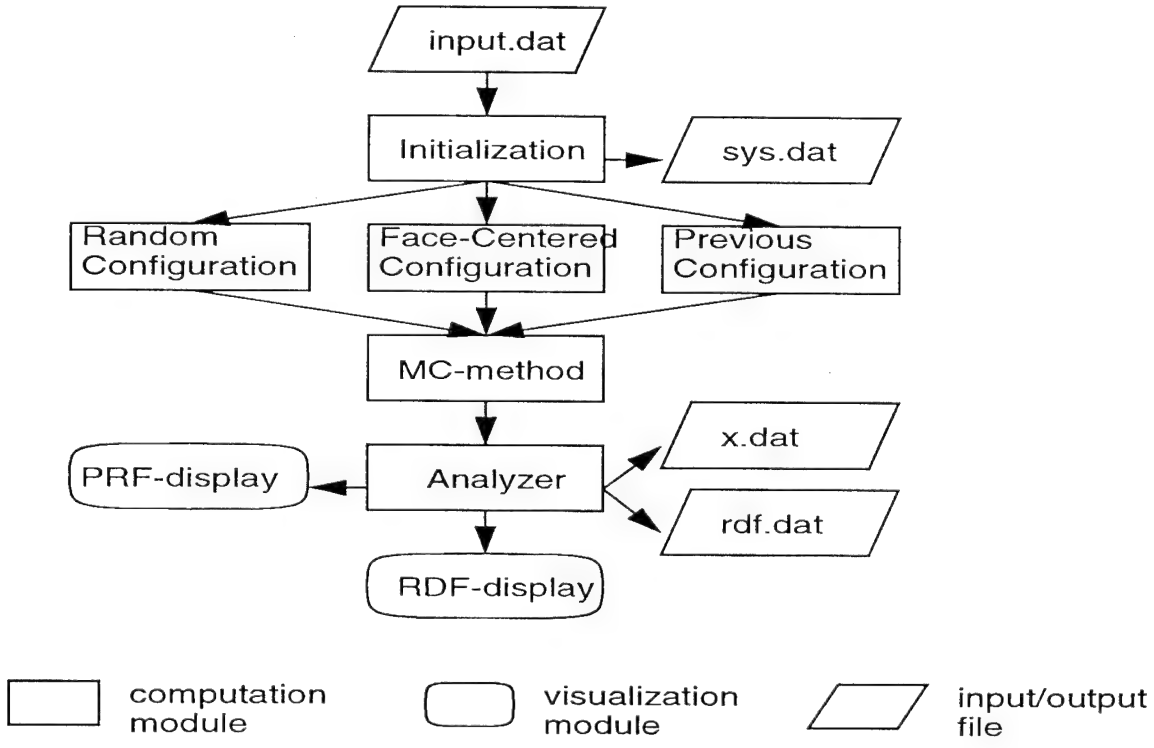

Figure 4: The application structure of colloidal dispersions

- The *initialization* module reads all parameters of the colloidal system from file 'input.dat', finishes the computation for initialization, and activates the proper configuration module. The system parameters are, for example, the number, diameter, and weight of particles, potential, pressure, and the number of iterations. Parts of system's log are stored in file "sys.dat".

- Because of the different characteristics of the three initial spatial configurations, we separate them into three different modules. These modules include 1) *randomly-generated configuration* module that is based on a good random number generator, 2) *face-centered configuration*, 3) and the *previous-running configuration* module that is based on file access. These three modules require nested-loop iterations. The activity of one module is determined by the *initialization* module at runtime.

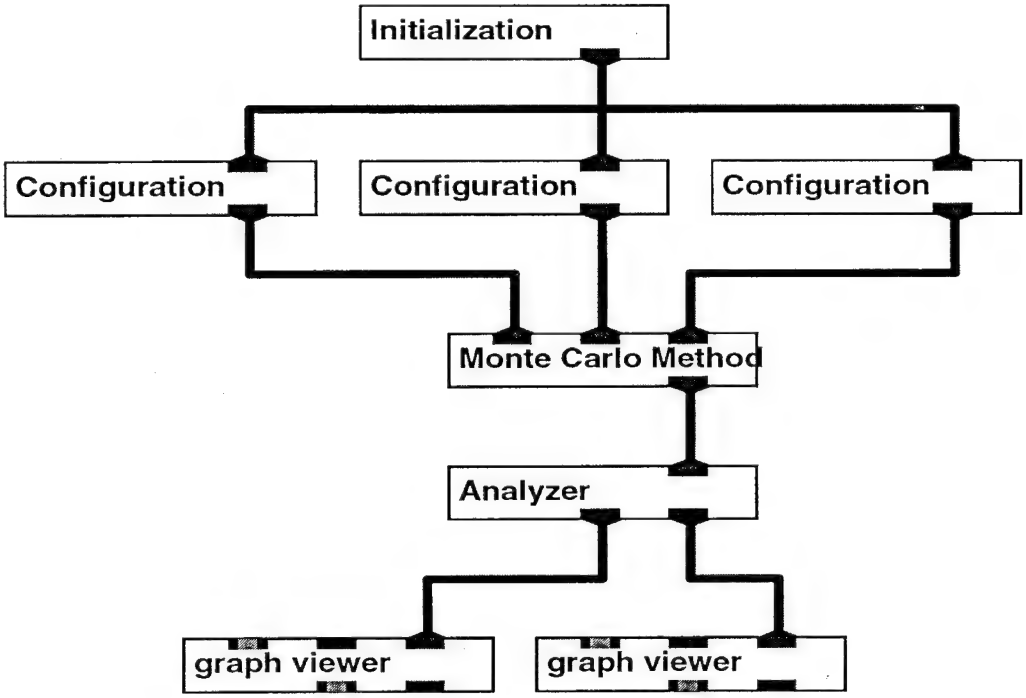

Figure 5: The application structure represented by AVS's Network Editor

- The major computation part of this application is concentrated on the *MC-method* module which implements the Monte Carlo simulation. For each iteration that represents a spatial configuration, the interparticle forces must be calculated for all particle pairs. Such a computation, called *N-body problem*, is the most time-consuming task in this application.

- The result of *MC-method* module is sent to the *analyzer* module. The numerical data of radial distribution function and execution time is forwarded to the *RDF-display* module and the *PRF-display* (PRF is PeRFormance for short) module, separately, for on-line visualization. One copy of radial distribution function and spatial configuration will be backed up in 'rdf.dat' and 'x.dat' files, respectively.

5.2 Task Characteristics and Computation Models

There are eight tasks in this application. Table 5.2 shows the best computing model to implement each task.

The detailed functional parallelism approach for Monte Carlo method is demonstrated in this section. The implementation of Monte Carlo method, based on the host-node program-

Table 2: The characteristics of tasks in Colloidal Dispersions

Task Name	Characteristics	Input data size	Number of Processors	Computational Model	Natural Support Hardware
Initialization	file access	-	1	sequential	SISD
Random Configuration	nested-loop iterations random number generation	32	32	data parallel	SIMD
Face-Centered Configuration	nested-loop iterations	32	32	data parallel	SIMD
Previous Configuration	file access nested-loop iterations	32	32	data parallel	SIMD
MC-method	nested-loop iterations N-body computation	32	32	functional parallel	MIMD
Analyzer	nested-loop iterations	128	128	data parallel	SIMD
PRF-display	visualization	-	1	graphical	SISD
RDF-display	visualization	-	1	graphical	SISD

ming model, is programmed in C and CMMD library and is executed on the CM-5. The host program governs the correction of particle positions, the communication among the node programs, and the calculation of radial distribution function. The node program is constructed according to the Metropolis Monte Carlo method. However, each node program calculates and controls the movement of only one target particle. In our study, there are 32 particles in the system. As a result, 32 processors are involved in this application.

At the outset, the host program broadcasts the initial configuration to all the node programs. Each node program starts with the same configuration and then moves its own target particle. A particle movement plus its evaluation is called one iteration. After each node program proceeds a certain number of iterations, the host program collects the position of particles from each node program, screens out the overlap cases, and distributes the correction index to every node program. The node programs update the configuration by broadcasting their target particle position. The new configuration is used for executing the next step. The node programs end when the total number of iterations reaches a pre-defined constant. Finally, the host program calculates the radial distribution function and prints it out.

There are several different communication/synchronization patterns among the host program and the node programs. Below is a brief description of these mechanisms:

- **point-to-point communication between host and node**: This function is used to pass, for example, the number of iterations and the execution time, from the node program to the host program. The point-to-point communication between host and node is implemented by **CMMD_send()**, and **CMMD_receive()**.

- **group communication among host and nodes**: There are three types of group communication among host and nodes.

  1. Concatenate data (the positions of each particle) amongst nodes and from nodes to host. This function is implemented by **CMMD_concat_elements_to_host()**, and **CMMD_gather_from_nodes()**.
  2. Broadcast host data (the correct particle position) to all nodes. This function is implemented by **CMMD_distrib_to_nodes()**, and **CMMD_receive_element_from_host()**.
  3. Perform reduce operations from the nodes to the host. This function is implemented by **CMMD_reduce_to_host_double()**, and **CMMD_reduce_from_nodes_double()**.

- **group communication among nodes**

  This function is used to update the local configuration for the next iteration. The group communication among nodes is implemented by CMMD_concat_with_nodes().

- **synchronization among nodes**

  This function is used to synchronize all the node programs before executing them. The group communication among nodes is implemented by CMMD_sync_with_nodes().

6 Conclusions

In this report, we have surveyed the current trends in computer hardware, computer software, network technology and application characteristics. The results of this survey showed the important role of these technologies to develop the virtual computing environment. We also presented an approach for creating a Virtual Computing Environment that can efficiently utilize multiple programming paradigms, programming languages, computer architectures, and physical machines for solving complex applications. These applications have complicated characteristics in computation, storage, visualization, heterogeneity, parallelism, communication, and synchronization, that exceed the capability of a single programming paradigm, programming language, computer architecture, and physical machine.

Our approach can be implemented by three phases: specification phase, configuration phase, and execution phase. For each phase, we identified the basic requirements and available software support. Furthermore, we used a scientific application, colloidal dispersion, to demonstrate our concept.

Further research needs to address the following issues:

- Identify the tasks that should be added to improve the capability of AVS to describe/develop the virtual machine applications.
- Develop a method to parse the file generated by AVS so that it can be used as an input to the configuration layer of the virtual computing environment.
- Identify the tasks that should be added to the ISIS resource manager so it can be used in the proposed virtual computing environment.
- Determine what rules should be used to select particular implementations of tasks.
- Develop a library of tasks that can be used to develop large complex applications. There are several issues that need to be addressed in developing such a library. For example, we need to determine how to implement such a library? How should the interface be specified?

- Develop a set of virtual machine applications (multi-target tracking, and colloidal dispersions) to test, benchmark and evaluate our approach to implement the proposed virtual computing environment.

References

[1] Angel L. DeCegama, *Parallel Processing Architectures and VLSI Hardware, Vol.1*, Prentice-Hall, Inc., 1989.

[2] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[3] H. Siegel and L. Siegel, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, vol. 30, pp. 934–947, December 1981.

[4] Kai Hwang, "Advanced Parallel Processing with Supercomputer Architectures," *Proceedings of the IEEE*, vol. 75, pp. 1348–1379, October 1987.

[5] Doreen Y. Cheng, "A Survey of Parallel Programming Languages and Tools," *RND-93-005, NAS Ames Research Center*, March 1993.

[6] R. H. Perrott, "Parallel Language Developments in Europe: An Overview," *Concurrency: Practice and Experience*, vol. 4(8), pp. 589–617, December 1992.

[7] Charles W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, pp. 131–182, June 1992.

[8] Ruben Prieto-Diaz and James M. Neighbors, "Module Interconnection Languages," *The Journal of Systems and Software*, vol. 6, pp. 307–334, 1986.

[9] Craig Upson, Thomas Faulhaber Jr., David Kamins, David LaidLaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications*, pp. 30–42, 1989.

[10] M. R. Barbacci, C. B. Weinstock, and J. M. Wing, "Durra: Language Support for Large-Grained Parallelism," *Parallel Processing and Applications, ed. E. Chiricozzi and A. Damico, North-Holland*, pp. 371–379, 1988.

[11] Jeff Kramer, "Configuration Programming - A Framework for the Development of Distributable Systems," *IEEE*, pp. 374–384, 1990.

[12] James M. Neighbors, "The Darco Approach to Constructing Software from Reusable Components," *IEEE Trans. on Software Engineering*, vol. SE-10, pp. 564–574, September 1984.

[13] Roberto Bisiani and Alessandro Forin, "Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems," *ACM*, pp. 21–30, 1987.

[14] Parasoft Corporation, "Express Reference Manual," 1988.

[15] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," 1991.

[16] Adam Beguelin, Jack J. Dongarra, G. A. Geist, Robert Manchek, and V. S. Sunderam, "Graphical Development Tools for Network-Based Concurrent Supercomputing," *Proceedings of ACM Supercomputing*, pp. 435–444, 1991.

[17] Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, , and Min-You Wu, "Fortran D Language Specifications," *Northeast Parallel Architectures Center, Syracuse University*, 1990.

[18] K. Birman, R. Cooper, T. Joseph, K. Kane, and F. Schmuck, "The ISIS System Manual".

[19] S. Ahuja, N. Crriero, and D. Gelernter, "Linda with Friends," *IEEE Computer*, vol. 19, pp. 26–34, August 1986.

[20] Andrew S. Grimshaw, "An Introduction to Parallel Object-Oriented Programming with Mentat," *TR-91-07, Department of Computer Science, University of Virginia*, 1991.

[21] Patrick T. Homer and Richard D. Schlichting, "Supporting Heterogeneity and Distribution in the Numerical Propulsion System Simulation Project," *Proc. of the Second Int. Symposium on High Performance Distributed Computing*, pp. 187–195, July 20-23 1993.

[22] Argonne National Laboratory, "Acquisition of a Highly Parallel Research Computer," February 1993.

[23] Richard L. Phillips, "Distributed Visualization at Los Alamos National Laboratory," *IEEE Computer*, pp. 70–77, August 1989.

[24] Brian Cabral and Carol L. Hunter, "Visualization Tools at Lawrence Livermore National Laboratory," *IEEE Computer*, pp. 77–84, August 1989.

[25] Robert B. Haber, "Scientific Visualization and the Rivers Project at the National Center for Supercomputing Applications," *IEEE Computer*, pp. 84–89, August 1989.

[26] T. F. La Porta and M. Schwartz, "Architectures, Features, and Implementation of High-Speed Transport Protocols," *IEEE Network Magazine*, pp. 14–22, May 1991.

[27] A. N. Netravali, W. D. Roome, and K. Sabnani, "Design and Implementation of a High-Speed Transport Protocol," *IEEE Trans. on Communications*, November 1990.

[28] N. Jain, M. Schwartz, and T. R. Bashkow, "Transport Protocol Processing at GBPS Rates," *Proceedings of the SIGCOMM Symposium on Communications Architecture and Protocols*, pp. 188–198, August 1990.

[29] Victor L. Peterson, John Kim, Terry L. Holst, George S. Deiwert, David M. Cooper, Andwer B. Watson, and F. Ron Bailey, "Supercomputer Requirements for Selected Disciplines Important to Aerospace," *Proceedings of the IEEE*, vol. 77, pp. 1038–1055, July 1989.

[30] Henry Fuchs, Marc Levoy, and Stephen M. Pizer, "Interactive Visualization of 3D Medical Data," *IEEE Computer*, pp. 46–51, August 1989.

[31] Charles C. Weems, "Architectural Requirements of Image Understanding with Respect to Parallel Processing," *Proceedings of the IEEE*, vol. 79, pp. 537–547, April 1991.

[32] Charles Weems, Edward Riseman, and Allen Hanson, "The DARPA Image Understanding Benchmark for Parallel Computers," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 1–24, 1991.

[33] Kai Hwang, *Advanced Computer Architecture with Parallel Programming*, McGraw-Hill, Inc., 1993.

[34] Geoffrey C. Fox, "The Architecture of Problems and Portable Parallel Software Systems," *Northeast Parallel Architectures Center, Syracuse University*, 1990.

[35] Ira Pramanick, "Distributed Computing Solutions to the All-Pairs Shortest Path Problem," *Proceedings of the Second Int. Symposium on High Performance Distributed Computing*, pp. 196–203, July 20-23 1993.

[36] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, Inc., 1988.

A Virtual Machine Programming Model For High-Performance Computing

H. J. Siegel and H. G. Dietz
Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285
April 21, 1994

Abstract

As parallel processing has developed, the execution models used with different systems have not converged. One is tempted to view this divergence as a sign of the field's immaturity; however, we suggest that the use of a range of architectural designs, and hence of various execution models, is primarily driven by the fact that the relative performance of different system designs is highly application dependent. Even individual functions within a parallel program often exhibit strong preferences for different system structures, and machines with the ideal structures may all be available within a single heterogeneous network. There is also the complication that, although a particular application might execute fastest when running by itself on one system, the best turnaround time might result from running the program on a different system that is less heavily loaded at the time the job is submitted.

For these reasons, it is vital that programs be highly portable. However, portability usually sacrifices execution speed - programs are ported to new machines by simulating the intended target machine's features (e.g., a MIMD simulating a SIMD). Instead, we propose that portability, and ease of program development, be achieved by presenting the programmer with an abstract *programming model* from which programs can be mechanically transformed into effective code for any of a wide range of target machine *execution models*. This report summarizes the progress that we have made toward these goals with the seed money we received from Rome and the research we propose to do for future work.

1 Scope Of Work

For the work performed under Rome Laboratory award number F30602-92-C-0150, we have focussed on the most basic issues that we identified in the original white paper. We concentrated on work that would help us define a plan for the research and development needed to make the virtual machine programming model a reality.

These issues center on the concept that a virtual machine programming model must be a complete, self-consistent, view of programming that is independent of the type of hardware used for execution. It should even be possible to debug one's program knowing **only** the programming model - without knowing the execution model used by the underlying supercomputer (or network of computers). This can be accomplished easily by picking an execution model and having every other machine simulate that (e.g., MIMD executing SIMD code by inserting a synchronization after each operation), but simulation sacrifices execution speed.

Thus, our research has been based on the premise that execution speed can be achieved by compiling programs directly into a native execution model for each target machine. Further, we want to use all the features of each target machine without making the features visible to the programmer, i.e., without making the program machine dependent and non-portable.

The wide variety of parallel execution models currently in use reflects application-dependent performance, and we want a single programming model to be able to target any execution model so that the best performance can be achieved. Because the handling of small-scale parallelism is relatively well understood, our initial work has emphasized the two most common types of large-scale parallelism:

- Multiple Instruction stream, Multiple Data stream (MIMD).
- Single Instruction stream, Multiple Data stream (SIMD).

In addition, we do not want to require the user to decide which execution model is best. Instead, the system should automatically minimize execution time for each user program by

- Predicting performance for the program under each combination of execution model and target machine using both static (e.g., expected execution costs) and dynamic (e.g., load average) information.
- Picking the target(s) where the program has the shortest expected execution time.
- Causing the program to run there.

Although there is still much work to be done, the initial Rome funding has allowed us to make significant progress toward this goal.

The following sections briefly discuss our major accomplishments. This report refers to, and includes, a list of the papers we published that were supported in part by this Rome contract. The interested reader is referred to these publications for details.

The initial funding was sufficient for us to construct a restricted implementation of a software system with the functionality described above. This system, AHS, is presented in Section 2. Section 3 describes the research we conducted with our initial funding from Rome toward enhancing the capabilities of the current AHS prototype. Also in Section 3, we discuss the future research we propose to do with follow-on funding from Rome so that the AHS concepts can be extended to become a working virtual machine programming model.

2 AHS: Automatic Heterogeneous Supercomputing

To more fully understand the problems inherent in implementing a virtual machine programming model for high-performance computing, and to evaluate the quality of proposed solutions, we felt it was vital to develop a prototype software system. This prototype, AHS, was designed and built at Purdue specifically as a testbed for the work funded by this Rome contract.

Although AHS is currently relatively crude, we believe that it is the first software system to implement all the fundamental properties of a virtual machine programming model:

- AHS allows users to write their programs using a single programming model, without any reference to or understanding of the target machine.
- Many execution models and target machines are supported.
- AHS performs flow-analysis based static cost estimation, and compiles each user program into an executable script.
- When executed, the AHS-generated script considers dynamic loads and then automatically distributes, compiles, and runs the program on the target(s) expected to minimize execution time.

The complete system is discussed in detail in conference paper [4] and technical report [1]. However, here it is useful to note a few of the more significant features.

Currently, the only programming model supported by AHS is an explicitly parallel dialect of C called MIMDC. This language uses a single-program multiple-data (SPMD) programming

model supporting an arbitrary number of virtual processors. Processes communicate using shared memory references; synchronization across all processors is supported by a fast barrier synchronization construct, although processes can also use shared memory semaphores to synchronize. In addition to MIMDC, AHS will eventually allow users to choose from a wide variety of programming models supported by explicitly parallel dialects of C and by automatically parallelized Fortran (Fortran-P).

Although the MIMDC programming model is that of a shared-memory MIMD, the target systems currently supported by AHS range from a message-passing SIMD supercomputer to a network of UNIX workstations. The range of targets supported could be increased. Presently, there are a variety of execution models implemented for AHS:

- **MIMD emulator.** Runs on MasPar MP-1/MP-2 SIMD supercomputers.
- **UNIX pipes.** Runs on any uniprocessor or multiprocessor UNIX system.
- **UNIX shared file.** Runs on most uniprocessor or multiprocessor UNIX systems, usually faster than pipes.
- **UDP socket.** Runs across any network of uniprocessor or multiprocessor UNIX systems that support the UDP protocol on BSD sockets.

Of the above, all but the UDP socket execution model have been debugged to at least β-test level. We selected these models as top priority because they span the widest range of target machines with the fewest possible execution models, thus, we have already been able to test the system with over a dozen different target machines:

- A 16,384-processing element MasPar MP-1 SIMD supercomputer.
- A 4-processor Stardent Triton.
- A 4-processor Sun Sparc Server.
- A 2-processor Gould NP-1.
- A 2-processor VAX 11/780.
- A wide variety of UNIX workstations.
- A variety of 386 and 486 microcomputers running Linux.

With further funding, our research will progress and the set of execution models supported will increase.

Like PCCTS (the Purdue Compiler-Construction Tool Set - the compiler construction system that we used to construct many key components of AHS), we intend that AHS will be available as a full source code public domain release. However, the funding provided thus far did not allow us to prepare appropriate documentation, etc., to support a general release of AHS.

3 Ongoing Research Work

Although AHS already satisfies the basic requirements we set forth for implementing a virtual machine programming model, there are a number of fundamental research issues that the current system does not address. Supported in part by this Rome contract, we have done significant basic research toward these issues. In the most general terms, the research covers

- Language technology to allow the user to give information that will help the system select better execution model(s) and target(s).
- Compiler methods for transforming parallelism (e.g., MIMD→SIMD).
- Algorithm mapping studies aimed at improving our ability to predict machine performance on a given task.
- Methods for finding the best *hybrid* execution model and target choices (finding target machines and execution models for *portions* of programs, rather than programs).
- A first step at migrating a task from a SPMD machine to a SIMD machine to balance load or recover from a fault.

Toward development of new language technology to allow the user to help the system select better execution model(s) and target(s), we have examined a number of issues involving the types of language constructs and compiler pragmas that could be used. In essence, our studies to date have simply confirmed and slightly extended the results of previous work we had conducted along these lines. For example, the use of data layout directives in the sense proposed for Fortran-D or HPF appears to be counter-productive because (1) the directives specify layout of variables, but it is often best for the compiler to use a different layout for each set of values held by each variable, (2) the data layout directives are too low-level, and are thus machine dependent and not efficiently portable, and (3) many of the data layouts

that are most useful are not representable in Fortran-D or HPF (e.g., layouts that decompose arrays into subarrays with replicated overlap, as used for many computational fluid dynamics codes). Instead, we focus on annotations that directly provide the compiler with better information about legal program behavior. For example, languages should be able to indicate precisely where nondeterministic accesses (races) would not adversely affect program execution. Another area for future research is to move AHS from using the MIMDC language to using a language that allows more freedom for the compiler to transform code for different execution models.

An important general issue related to this is the determination of at what level of abstraction the language should be. This will be a balance among factors such as ease of programming, complexity of compiling efficient code for a variety of target machines, portability, maximizing user-supplied information that will aid compilation for a variety of targets, and preventing a user from forcing unnecessary precedence conditions or ineffective data layouts.

Supported by the Office of Naval Research, we have developed a number of techniques that allow MIMD code to be transformed for efficient execution on SIMD hardware. For example, this work has produced a MIMD environment which makes a SIMD MasPar MP-1 able to efficiently employ an execution model that provides a 16,384 processor shared memory MIMD. Conversely, earlier work (e.g., in code scheduling for barrier MIMD) provides the ability to efficiently execute SIMD code on machines whose native hardware appears to be MIMD. Thus, for the work supported by this Rome contract, we have been able to consider multiple execution models for each hardware target, i.e., both SIMD and MIMD execution models can be efficiently supported by the MasPar. Consequently, our research attempts to determine which execution model should be used by considering not only transformation of the programming model into the native execution model for each target, but also direct compiler transformation of programs into other execution models. Further research is needed to develop a theoretical foundation and to bring this new technology into AHS.

Our algorithm mapping work focuses on problems underlying the mapping of algorithms that would be written for the virtual machine onto different target parallel machines. Some algorithms simply work better on specific types of machines, and an alternative algorithm may work better on the particular machines you have available. In essence, we are attempting to determine how high-level and abstract algorithm specifications can be made, as was mentioned above. The ideal would be to have the algorithm specifications be general enough that the compilation system could automatically generate the variations on the algorithm that are best suited to the available execution models and target machines. Salient parameters we have studied include trade-offs among SIMD, MIMD, and mixed-mode execution, performance prediction, the impact of changes in machine and problem size, the scalability of an approach,

and the relationship between algorithm structure and the inter-processor communications network. Thus far, this work has used the case studies listed below to examine these problems for the SIMD MasPar MP-1, the MIMD nCUBE 2, and the mixed-mode SIMD/MIMD PASM prototype. With additional funding, these studies would continue, helping to guide the development of all other aspects of the system. The continuation of this work would also naturally lead to the definition of a set of library routines needed across the desired target machines.

The following algorithm studies were supported in part by this Rome contract. In conference paper [1], we demonstrate how parallel systems which differ only in the topology of their inter-processor network can require fundamentally different algorithmic approaches to perform the same task. The parallelization of the application of "cyclic reduction" is presented at various levels of detail in journal paper [1], conference paper [3], and Ph.D. thesis [1]. The parallelization of the application of solving multiple-quadratic forms is presented at various levels of detail in journal paper [2], conference paper [5], and technical report [2]. These two application studies explore the parameters mentioned in the previous paragraph by theoretical analyses coupled with and verified by experimentation on the MP-1, nCUBE 2, and PASM.

When the current version of AHS determines where and how a program should be executed to minimize expected execution time, it considers the entire program as an atomic entity. We have done considerable research work toward extending AHS to consider breaking programs into segments that may be executed using different execution models and target machines. Part of this research involves mechanizing estimation of the relative execution time of a data-parallel algorithm in an environment capable of both SIMD and SPMD modes of computation. The work we conducted on this subject, supported in part by this Rome contract involved both the PASM mixed-mode system and mixed-machine heterogeneous systems. Journal paper [3] and conference paper [2] present the basic framework for a technique that can be used in a mixed-mode machine. Conference paper [6] and Ph.D. thesis [2] show how this technique provides an excellent heuristic in a multiple-machine heterogeneous environment. Further research is needed to refine the framework developed thus far.

The technique starts with a data-parallel program written in a virtual machine model language and empirical information about instruction execution time characteristics on specific architectures. From this, it determines how the program should be decomposed into portions that could be executed using different execution models and target machines, and then determines the set of implementation choices that globally minimizes the expected execution time. We can conduct the research needed to fill in the details of the framework we have developed for this technique. A secondary goal of this study is to indicate language, algorithm, and machine characteristics that must be researched to learn how to provide the information needed to obtain an optimal assignment of parallel modes to program segments. It then can

be incorporated into AHS.

In conference paper [7], one part of a method to migrate a task dynamically from a SPMD machine to a SIMD machine is proposed. Such a migration may be desired for load balancing or in case of a fault. It is assumed for this initial work that the SIMD and SPMD machines differ only in their support of different modes of parallel execution, and that the program was coded in a language that is mode independent. The migration procedure does not require that the SPMD processing elements be at the same location in the SPMD program at the time of the migration. This work is directly applicable to mixed-mode hybrid SIMD/SPMD systems and part of the general problem of task migration in SIMD/SPMD mixed-machine heterogeneous systems.

In addition to the above, there are other related areas where research is needed. Experimental studies are needed to consider the impact of LAN traffic on inter-machine communication time, how often should machine load information be updated in the job manager, what user-specified information is truly useful, etc. The fundamental research issues that need to be pursued include: determining the level of abstraction for the programming language construct, new compiler technology, the inference of the data layout on different architectures, decomposition of a program for execution across multiple heterogeneous machines, and the migration of tasks between machines due to faults or load changes.

Publications Supported in Part by Rome Laboratory Under This Contract

Journal Papers:

[1] Gene Saghi, Howard Jay Siegel, and Jeffery L. Gray, "Predicting Performance and Selecting Modes of Parallelism: A Case Study Using Cyclic Reduction on Three Parallel Machines," *Journal of Parallel and Distributed Computing,* Vol. 19, No. 3, pp. 219-233, Nov. 1993.

[2] Mu-Cheng Wang, Wayne G. Nation, James B. Armstrong, Howard Jay Siegel, Shin-Dug Kim, Mark A. Nichols, and Michael Gherrity, "Multiple Quadratic Forms: A Case Study in the Design of Data-Parallel Algorithms," *Journal of Parallel and Distributed Computing,* Special Issue on Data-Parallel Algorithms and Programming, accepted and to appear in 1994.

[3] Daniel W. Watson, Howard Jay Siegel, John K. Antonio, Mark A. Nichols, and Mikhail J. Atallah, "A Block-Based Mode Selection Model for SIMD/SPMD Parallel Environments," *Journal of Parallel and Distributed Computing,* Special Issue on Heterogeneous Processing, accepted and to appear in 1994.

Conference Papers:

[1] Howard Jay Siegel, John K. Antonio, and Kathy J. Liszka, "Metrics for Metrics: Why It Is Difficult to Compare Interconnection Networks OR How Would You Compare an Alligator to an Armadillo?" *IEEE Proceedings of The New Frontiers: A Workshop on Future Directions of Massively Parallel Processing,* pp. 97-106, Oct. 1992.

[2] Daniel W. Watson, Howard Jay Siegel, John K. Antonio, Mark A. Nichols, and Mikhail J. Atallah, "A Framework for Compile-Time Selection of Parallel Modes in an SIMD/SPMD Heterogeneous Environment," *IEEE Proceedings of the 2nd Workshop on Heterogeneous Processing,* pp. 57-64, Apr. 1993.

[3] Gene Saghi, H. J. Siegel, and Jeffrey L. Gray, "Mapping onto Three Classes of Parallel Machines: A Case Study Using the Cyclic Reduction Algorithm," *IEEE Proceedings of the Seventh International Parallel Processing Symposium,* pp. 238-247, Apr. 1993.

[4] Henry G. Dietz, William E. Cohen, and Brian K. Grant, "Would You Run It Here... Or There? (AHS: Automatic Heterogeneous Supercomputing)," *Proceedings of the 1993 International Conference on Parallel Processing, Vol. II,* pp. 217-221, Aug. 1993.

[5] Mu-Cheng Wang, Wayne G. Nation, James B. Armstrong, Howard Jay Siegel, Shin-Dug Kim, Mark A. Nichols, and Michael Gherrity, "Multiple Quadratic Forms: A Case Study in the Design of Scalable Algorithms," *Proceedings of the 1993 International Conference on Parallel Processing, Vol. III,* pp. 37-46, Aug. 1993.

[6] Daniel W. Watson, John K. Antonio, Howard Jay Siegel, and Mikhail J. Atallah, "Static Program Decomposition Among Machines in an SIMD/SPMD Heterogeneous Environment with Non-Constant Mode Switching Costs," *Proceedings of the Third Workshop on Heterogeneous Computing,* accepted and to appear in Apr. 1994.

[7] James B. Armstrong, Howard Jay Siegel, William E. Cohen, Min Tan, Henry G. Dietz, and Jose A. B. Fortes, "Dynamic Task Migration From SPMD To SIMD Virtual Machines," *Proceedings of the 1994 International Conference on Parallel Processing,* accepted and to appear in Aug. 1994.

Technical Reports:

[1] Henry G. Dietz, William E. Cohen, and Brian K. Grant, "Would You Run It Here... Or There? (AHS: Automatic Heterogeneous Supercomputing)," Purdue University, School of Electrical Engineering, Technical Report No. TR-EE 93-5, Jan. 1993, 22 pp.

[2] Mu-Cheng Wang, Wayne G. Nation, James B. Armstrong, Howard Jay Siegel, Shin-Dug Kim, Mark A. Nichols, and Michael Gherrity, "Computing Multiple Quadratic Forms for a Minimum Variance Distortionless Response Adaptive Beamformer Using Parallelism: Analyses and Experiments," Purdue University, School of Electrical Engineering, Technical Report No. TR-EE 93-20, May 1993, 45 pp.

Ph.D. Theses:

[1] Gene Saghi (Advisor: Howard Jay Siegel), "Compiler, Fault Tolerance, and Performance Prediction Aspects of Reconfigurable Parallel Processing Systems," May 1993.

[2] Daniel W. Watson (Advisor: Howard Jay Siegel), "Compile-Time Selection of Parallel Modes in an SIMD/SPMD Heterogeneous Parallel Environment," Aug. 1993.

Dynamic Thresholds in Process Allocation

V.Bharghavan C.V. Ramamoorthy
CS Division, Department of EECS
University of California at Berkeley

R. Mittal
Department of CSE
Indian Institute of Technology at Madras

Abstract

We present an algorithm for allocating a dynamic network of processes on a dynamic network of multiprocessors and workstations communicating with a Client-Server model. Our effort has been to provide a real-time solution using only local information and computation. Although the problem of load sharing has been extensively researched, we believe that little work has been done in the context of this particular problem scenario. Among the important features of this algorithm is its adaptability, use of local computation only, and efficient distribution of load in real time. We expect this algorithm to be specifically useful in future Client-Server models in parallel systems that use multiple server threads to provide service.

1 Introduction

This paper proposes a process allocation algorithm for the popular Client-Server model in a network of processors, where Server processors could be multiprocessors. The intended network is expected to be a cluster of multiprocessors and workstations connected by a high-speed network. The processor graph thus looks like a hierarchy of processor clusters. Such environments are already available for general purpose use and are expected to become widespread soon. Such an environment poses a variation to the existing problem of load sharing. In existing models of Client-Server computation, the simplistic view of a service is one in which a process or thread is created for each client at the server site. However, with the evolution and widespread use of multiprocessors, it will be common to exploit parallelism and spawn multiple threads to do a single task. In other words, the simplistic view of the Client-Server model in the future will be a number of cooperating threads spawned in order to provide

service for a single Client. In such a scenario, we expect that two kinds of problems will need to be solved - which server to request for a service (a macro-level load balancing), and how to spawn threads locally in order to provide a service effectively (a micro-level load balancing). This problem is a variant of the existing one, and we have not seen much literature on this problem. Our effort solves this problem, keeping two most important constraints in mind - Locality of Information and Computation, and Speed in arriving at the thread allocation. Execution of a parallel computation on a network of processors involves two important phases - Task allocation, and Task Scheduling, both well known NP hard problems. Task allocation involves the mapping of processes onto processors. Task scheduling involves scheduling the execution of processes on each processor. In the case of static networks, we can do the Task Allocation and Task Scheduling before runtime. However, for any application of even moderate size, static networks are clearly a very restrictive model. Our focus has been to provide constructive and useful algorithms which will be actually used in real-time applications. Thus we consider dynamic networks of processes and processors. Consequently, Task allocation and scheduling must be done at run time. Since our targeted applications are real time, we have two important aspects to consider - a good solution, and a fast solution. Most of the work in the literature has tried to find the best mapping algorithm assuming a number of restrictions. Our focus is different. We want to see how efficiently we can map a process in real time, requiring as little information about the process or processor network as possible. Rather than trying to get as close to optimal as possible, introducing stringent restrictions (static network model, global knowledge of processor loads, process migration facility, etc), we want to see what we can do assuming as little as possible (dynamic network model, only local knowledge, no process migration during process execution, etc). We call our algorithm the Dynamic Threshold Algorithm since the task allocation among processors is governed by a Dynamic Threshold at each processor (Section 4).

2 Related Work

Task Allocation [2, 4, 8, 9, 12, 13, 14, 15, 16, 19] and Task Scheduling [6, 7, 10, 11, 20, 21, 22] in a network of Homogeneous or Heterogeneous processors is a thoroughly investigated problem. Real-time scheduling [17] is also a well researched topic. We are looking at the area of real-time Task Allocation in a network of processors, where the network of processes and processors can change with time. Our model of communication is Client-Server, and we assume the spawning of multiple threads to provide service at the Server site. Our dynamic load allocation is based on the concept of load balancing [6, 12]. Static Thresholds and Greedy

algorithms have been used in load balancing techniques. The Dynamic Thresholds differ from Static Thresholds in that they are generated from within the system and do not need to be imposed from outside by the user. We have combined Dynamic Thresholds with the Greedy technique [7] in order to improve the performance of the algorithm. For a comprehensive annotated bibliography on related work, see [5].

3 Class of Applications Under Consideration

We generalize the processor architecture from one in which multiprocessors interact, to one in which servers provide service by locally spawning threads in their neighborhoods, while Server-Client communication typically involves communication across the LAN or even WAN. Hence, request for service is typically a relatively long distance and infrequent communication, while cooperative communication between spawned threads to provide service is relatively short distance and frequent communication. In the Client-Server model, we have two kinds of processes: Clients, which request some service, and Servers, which provide the service. A Server provides a service to a client by creating one or more child processes (or threads - we use the term processes and threads interchangeably in the context of server threads that provide a service cooperatively), which cooperatively perform the tasks requested by the Client. The child processes may further spawn other processes. A hierarchy of processes (typically not very deep) is thus created, and processes within the hierarchy typically communicate with each other. Upon completion of the tasks, the child processes are destroyed, and the Server finishes up the processing of the Client request. From the Client-Server model, we make the following observations. Most of the communication occurs between processes that are related to the same computation. The Client makes a request to the Server, and after the computation is done, the Server returns to the Client. The Client-Server communication is typically long-distance, while the communication between processes that cooperatively solve the tasks is typically short-distance, where long-distance and short-distance refer to the communication distance between the processors on which the processes execute. Considering the general case, the Dynamic Threshold algorithm tries to optimize the case when processes created locally must communicate with each other frequently. While the Dynamic Threshold Algorithm is particularly suited for Client-Server model based applications, it is a general technique that can be used for the task allocation of any parallel computation.

4 Dynamic Threshold Algorithm

4.1 The Process and Processor Network

The Dynamic Threshold Algorithm is guided by the observations made about the typical application behavior, and also by the fact that the applications are real time. Each process p is assigned a Computation Weight *Cp(p)*, and two communicating processes p and q are assigned a Communication Weight *Cm(p,q)*. When a process p is created, it is provided its Computation Weight, and its Communication Weight with all communicating processes that are already alive. This means that at creation time, we need not predict the communication between p and processes which may be created at a later instant in time. If two processes communicate, the process that is created later will specify the Communication Weight between them. Each processor P is assigned a Processing Cost *Pc(P)*, Load *L(P)*, and a Threshold *T(P)*. The Processing Cost is the cost of processing a unit Weight of a process. Each link l in the network is assigned a Communication Cost *C(l)*. The Communication Cost is the cost of sending a unit Weight of messages across the link. The Communication Cost is very low for two processors in the same multiprocessor, while it is significantly higher for processors communicating across the LAN. The cost of communication between two processes in the same network is neglected. Initially, the Threshold in every processor is reset to 0. The Load at a processor is the sum of two components - The Computation component, and the Communication component. The Computation component at a processor is the weighted sum of the Computation Weight of every process allocated to the processor (weighted by the Processing Cost of the processor). The Communication component of a processor is the sum of the Total Communication Weight of every process executing on the processor, where the Total Communication Weight of a process is the weighted sum of the Communication Weight of the process and each of its neighbors not executing on the same processor (weighted by the Communication Cost of the link).

$L(P) = \Sigma(Cp(p) \forall p \in P) * Pc(P) +$
$\Sigma(\forall p \in P(\Sigma(Cm(p,q) * C(l))$
$\forall q$ not allocated on P, that p communicates with).

A Processor can send/receive query messages to/from its neighbor processors only. A Processor can query a neighbor processor about its Load, and the processes that are executing on the neighbor. We define the neighborhood of a processor P, *N(P)*, as follows.

$N(P) = Q : Q$ is a neighbor of P.

4.2 Process Creation

When a process $p1$ is created by a process p at a processor P, if the total Load at the processor P after $p1$ is allocated to P is not greater than the Threshold, then $p1$ is allocated to P.

$L(P) + Cp(p1) * Pc(P) + (\Sigma Cm(q,p1) * C(l) : q$ not running on $P) \leq T(P) \Rightarrow$ allocate $p1$ to P.

However, if the total Load after allocation of $p1$ will be greater than the Threshold at P, then we try to allocate the newly created process on some other processor in the neighborhood of P. Since we have access only to local information at P, and since the mapping is to be done in real time, we consider only a distribution in the neighborhood. In general, the processor to which the process is allocated does not initiate another distribution in its neighborhood. We deal with the case of cascaded allocation calls in Section 6.4. The process distribution is done as follows. The processor P queries every neighbor processor for its Load and the processes allocated to it. Upon receipt of status messages from all the neighbors, $p1$ is allocated to a processor such that the maximum Load in the neighborhood is minimized. If the process is allocated to $P2$, then the Load at $P2$ is

$L(P2) + Cp(p1) * Pc(P2) + (\Sigma Cm(q,p1) * C(l)$ for q not running on $P2)$
and the Load at every other processor $P1$ in the neighborhood of P is
$L(P1) + (\Sigma(Cm(q,p1) * C(l)$ for q running on $P1)$

We allocate the process $p1$ to a processor $P2$, such that the maximum Load in the neighborhood is minimized. If there is more than one processor to which the process could be allocated, then one among the set of processors is chosen. This choice could be random, or be based on some heuristics, as described in Section 6.1. In the allocation of a process p created at processor P, we have considered the communication in only the neighborhood of P. We have not updated the Load for communication with processes that are allocated to processors which are not in the neighborhood of P. We justify this decision based on our observation that most inter-process communication is short-distance, and also that all allocation decisions must be taken in real time. In many cases, this justification is not valid. For example, if this algorithm is applied to situations that approximately behave like the Client-Server model and still have a non-negligible communication between non-neighboring processors, then we cannot ignore the non-neighbor interprocessor communication. We now consider an extension to this allocation strategy. When a process p, which is allocated to a processor P, wants to communicate with a process $p1$, which is allocated to a processor $P1$, and $P1$ is not a neighbor of P, then the Load of P and $P1$ have to be updated (since this communication was not considered before). Besides, all the intermediate processors which are involved in the

communication of p and $p1$ also have their loads updated (this is due to the the routing done at the intermediate processors). If the Load at any processor as a result of this update is now greater than its Threshold, then the Threshold is set to the value of the Load. We will show in the Section 5 that the property of the Threshold is still maintained. The overhead in the process creation is n messages if there is no process distribution or $3n$ messages if there is a process distribution, where n is the degree of the processor at which the process is created (For details, see Section 5.4).

4.3 Process Deletion

When a process p is destroyed (exits or is killed) at a processor P, in the simple case when we ignore non-neighbor inter-processor communication, a message is sent to each neighboring processor of P which has allocated to it, a neighboring process of p. If we consider the more general case, a message is sent to every processor to which is allocated a neighboring process of p. The load at the processors is reduced by the communication cost due to p (and at P, also the processing cost due to p). If the Load of any processor was equal to its Threshold, the Threshold is reduced to the current Load value.

5 Properties of the Dynamic Threshold Algorithm

In the Dynamic Threshold algorithm, the most important concept is the generation of the Dynamic Threshold. It is useful to find out what the Dynamic Threshold signifies. Based on the algorithm, we observe that the Dynamic Threshold satisfies the following properties.

Property1. For every processor P, $Load(P) \leq Threshold(P)$. Initially, for each processor, $Load(P) = Threshold(P) = 0$. If a process is created at P, it is assigned to P only if $Load(P) \leq Threshold(P)$ after the allocation of the process. If a process is allocated to P by the load distribution algorithm then the Threshold is recomputed for P. The new Threshold is the maximum Load in the neighborhood of a neighbor of P. Therefore, $Load(P) \leq Threshold(P)$. After a process dies at P, the Threshold is either not changed, or is reduced by the same value as the Load at P. The way the Dynamic Threshold algorithm works is, whenever the load is changed, if it is greater than the Threshold, the Threshold is increased to at least the value of the load. Therefore, at all instants, the load of a processor is less than or equal to the Threshold of the processor.

Property2. The Threshold need not be the same for all processors at any time, or the same for one processor at all times. This is clear from the very nature of the algorithm. Dy-

namic Threshold is recomputed for a neighborhood, on-demand. So it can vary spatially, and temporally.

Property3. Every individual load distribution is locally optimal. The time taken for a computation is the maximum time taken by any processor participating in the computation. So we want to minimize the maximum Load in the network. Since we are only considering a neighborhood at any time, we want to minimize the maximum Load in the neighborhood. The process distribution algorithm allocates the process such that the maximum Load in the neighborhood is minimized. Hence it is locally optimal for each load distribution.

Property4. The allocation of a process p created at a processor P incurs a message overhead of n or $3n$ messages, where n is the degree of P, and a time overhead of T or $3T$, where T is the message transit time on a link. In case p is assigned to P, O(n) messages need to be sent to the neighbors of P, to update the load of the processors in the neighborhood. Messages are sent to only those processors whose load is changed due to the creation of p. In the case of process distribution, P queries each of its neighbors for its Load and the processes allocated to it. This requires n messages. Each of the neighbor processors sends its Load and processes. This requires the receipt of n messages. P then allocates p in the neighborhood, and possibly recomputes the Threshold for the neighborhood. Updating the load and threshold in the neighborhood requires n messages. If the time for a receive or send function is neglected with respect to the actual message transit time, the overhead is thrice the message transit time. Thus, if the message transit time in the link is T, the overhead is approximately $3T$. For the case of no process distribution, the time overhead is T.

Property5. Intuitively, the comparison of the Thresholds across processors at any instant indicates the spatial distribution of the load in the processor network, while the comparison of the Threshold of one processor along the time axis indicates the temporal distribution of the load in the neighborhood of the processor.

6 Comments on the Dynamic Threshold Algorithm

6.1 Optimal Choice of a Processor for Process Allocation

The primary goal of process distribution in the Dynamic Threshold Algorithm is to reduce the maximum load in the neighborhood. When allocating a new process in a neighborhood, several processors could be allocated the process and all result in the minimum maximum-Load in the neighborhood. Which of these processors should be chosen? If the process was created at processor P, and P is in the set of acceptable processors to which the process can

be allocated, we allocate the process to P. This will reduce the overhead due to inter-processor allocation. If P is not in the acceptable set, there are several heuristics that could be applied. The process could be allocated to that processor which maximizes the minimum load and minimizes the maximum load in the neighborhood. This will reduce the variance of load in the neighborhood. The process could be allocated to the processor with maximum degree, so that the process graph quickly spreads over the locality of the processor graph. For the same reason, the processor furthest away from the grandparent of the process could be chosen. The basic idea of these techniques is to quickly spread the process graph over the processor graph as the process hierarchy deepens. If response time is critical, we could just choose a random processor from among the set of acceptable processors. There are several ways of choosing one among the several possible processors, each of which has some intuitive appeal. We have not specified in this algorithm which choice to make. Different choices could prove to be best for different applications.

6.2 Worst Case Behavior

When a process dies at a processor, if the load at the processor prior to its death was equal to the Threshold, then we decrease both the load and the Threshold of the processor by the weight of the process. We could recompute the Threshold in the neighborhood again, but we don't do the recomputation in order to save time. One problem in not doing the recomputation is that the worst case behavior of this algorithm is very poor. Consider the following sequence of events. A processor P has a Load L, and its Threshold is also L. A neighboring processor $P1$ has a Load $L1 (L1 < L)$, and its Threshold is L. Now, all the processes in P terminate. The Load and Threshold of P is then 0. Now, all the processes in $P1$ terminate. The Load of $P1$ is 0, but its Threshold is still at L. This means, that if processes are now allocated or created at $P1$, till the Load reaches L, $P1$ makes no attempt to distribute the processes, even though its neighbor is free. The worst case behavior is thus one of no process distribution. One way out of this kind of worst case behavior is to recompute the Threshold in the neighborhood every time a process dies at a processor whose Load is at its Threshold. This is clearly an expensive way to improve the worst case behavior. A better way of improving the worst case behavior is adopting the Double-Sided Dynamic Threshold algorithm.

6.3 Double-Sided Dynamic Threshold Algorithm

In the Double-Sided Dynamic Threshold algorithm, we have two Thresholds, the Upper-Threshold, and the Lower-Threshold. Recomputation of Thresholds in the Dynamic Threshold

Algorithm now involves recomputing both the Upper and Lower Thresholds. However, the number of messages required to be sent across processors remains the same, since a processor can update both Thresholds of each neighbor using a single message. The case of process creation is the same as before. When a process dies at the processor, as before, the Upper Threshold may be lowered. Now, if the Upper Threshold is equal to the Lower Threshold, then the Thresholds are recomputed in the neighborhood. Let us now re-examine the sequence of events that caused the worst case behavior. P is a processor whose Load and Upper Threshold are both L. $P1$ is a neighbor of P, and its Load is $L1$. The Upper Threshold of $P1$ is L, and $L1$ is less than L. The Lower Threshold of P is l, and the Lower Threshold of $P1$ is $l1$. Now, all processes in P terminate. When the Threshold of P reaches l, then the Thresholds are recomputed in the neighborhood. So, only after the Load of P reaches the locally minimum Load, will the recomputations start taking place after every process dies at P. Since the Upper Thresholds are simultaneously, the Threshold follows the maximum Load in the neighborhood, and the worst case behavior is considerably improved. This algorithm still has to bear the burden of the recomputation of Thresholds for every process death once the Load of the processor is locally minimum. However, it partially alleviates the worst case behavior problem encountered before. Besides it has the pleasing property that the Upper Threshold and the Lower Threshold at a processor define the upper and lower bounds of the Load at the processor at every instant.

6.4 Cascaded Process Distribution

It may so happen that in some applications, all the processes are being generated by one parent process. We consider only local distribution of a process in the Dynamic Threshold algorithm. Thus the Load distribution is only in the neighborhood of the processor to which the parent process is allocated, even though processors a distance two or more away could be idle. Again, this problem has no 'best' solution. If we want a processor to have only local information, and also do the allocation in real time, we have to sacrifice some performance. Trying to repeatedly distribute a process so long as the Threshold the assigned processor is lesser than the new Load (if the new process were to be allocated on the processor) is clearly not feasible. We use a compromise solution (which is clumsy) by specifying a 'tolerance limit' for a processor, which is the maximum Load difference between two neighboring processors that is allowed. If the Load difference exceeds this tolerance limit, then processes assigned to the higher loaded processor may be redistributed by this processor in its neighborhood. This imposes a certain fan-out of the processes, but the specification of the tolerance limit is up to the user. We do not require the latest information to be maintained about the Load of

every neighboring processor at a processor. Periodically, the Load of all neighbors could be examined by a processor. This solution is clearly not elegant. This is however, an unavoidable compromise that we have to make in restricting the information at a processor to be local.

6.5 Overheads

As described in Property 4 in Section 5, the message overhead in process creation is n or 3n, and the time overhead is T or 3T, where n and T are respectively, the degree of the processor at which the process is created, and message transit time. When a process with a degree m is destroyed, a maximum of m messages are sent to other processors (each of the neighboring processes may be allocated to a different processor). The overhead of n messages on process creation and O(m) messages on process termination is the minimum which *has* to be incurred. This is because every processor has to be informed if its load is changing. Apart from this, the Dynamic Threshold based process distribution requires an overhead of 2n messages. Since our algorithm incurs only the minimum overhead on process termination, and minimum overhead on process creation most of the time (when there is no process distribution), and since the overhead in process distribution is very low, we expect the real time response of our algorithm to be acceptable. We have adopted the demand-driven approach for getting the status information about the neighborhood at a processor. A processor queries its neighbors for their status only when it has to distribute a process. An alternative method would be to periodically query the status of the neighbors, and maintain the status of the neighborhood at every processor. We have rejected this approach because this might lead to inconsistencies in the status information if two simultaneous requests are made at the same time in two neighboring processors. In many cases, the latter approach might however, lead to better performance. The Dynamic Threshold algorithm does not require global information, static networks, identical processor or link speeds, or process migration. It does require an estimate of the computation and communication costs of the processes. It does not aim to provide a near optimal allocation, but only a 'good' allocation as fast as possible, placing as few restrictions as possible.

7 Greedy Dynamic Threshold Algorithm

In the Client-Server model, many clients may request some service from the server, possibly concurrently. As in the model described before, the Server creates a hierarchy of processes to service each client request. Processes in the same hierarchy interact with each other and provide the one instance of the service, but processes belonging to different hierarchies, do

not interact with each other. We make use of this observation in the distribution of the processes in the processor network. The root of the hierarchy of an instance of service (we assume, without loss of generality, that there is such a root) can be allocated to a processor whose neighborhood is 'relatively unloaded' compared to the neighborhood of the Server. In the neighborhood of this remote processor, the hierarchy of processes is created, interact, do some computation, and return the result to the server. Different instances of the service could thus be spread out in the processor network. In this section, we propose the combination of the Dynamic Threshold and the Greedy algorithms to achieve a better distribution of processes in the network of processors. Every processor has a sequence of processors on which it tries to distribute processes. A tag bit is associated with each processor in the sequence. The sequence of processors depends on the network of processors. The sequence contains processors belonging to different clusters, in ascending order of distance. The exact algorithm governing the generation of processor sequences for each processor is not dictated by this algorithm. We assume that there exists such a sequence at each processor. When a Server gets a request for a service, if it has currently created another instance of the service in its neighborhood, it finds another processor on which to create the new instance of the service. This processor is found in a greedy manner. Basically, the processor checks if each of the processors in the sequence has an instance of some service currently allocated to it. If there is any available processor, a new service instance is created in its neighborhood, and the tag field is updated. If there is no such processor, the new instance of the service is also created in the neighborhood of the Server itself. Whenever a service is created or terminated at a processor, it updates the tag bits in each of the sequences where it occurs. Since the creation and termination of a service is much rarer as compared to the communication within a service, we accept the overhead in updating the tag bits in all sequences where a processor occurs. Once the root of the service is created in a processor, the hierarchy of processors is allocated by the Dynamic Threshold Algorithm .

8 Conclusion

Our focus has been real time process allocation of dynamic process networks on dynamic processor networks, specific to the Client-Server model in multiprocessor and Workstation networks. We have proposed a Dynamic Threshold based algorithm that produces a good process allocation strategy involving little overhead in message transfer. The essence of the Dynamic Threshold is that the Threshold is generated by the system from within, in response to the external load, and is not imposed by the user. So it is more responsive to load fluctua-

tions. The drawback is that since only local information is used, wide fluctuations in processor loads are possible if the spatial distribution of the process generation is highly uneven. We have addressed this issue at two levels. At the intra-hierarchical level, we have incorporated cascaded distribution, and at the inter-hierarchical level, we have proposed a Greedy Dynamic Threshold Algorithm. We have worked within the constraints imposed by the real time requirements of dynamic process allocation. Our future work will test the efficiency of the Dynamic Threshold algorithm in critical real-time problems.

References

[1] Andrews, G.R. Paradigms for Process Interaction in Distributed Systems, *ACM Computing Surveys*, 23, 1, 1991, pp. 49 - 90.

[2] Berman, F. and Snyder, L. On Mapping Parallel Algorithms into Parallel Architectures, *Journal of Parallel and Distributed Computing*, 4, 1987, pp. 439 - 458.

[3] Bokhari, S. A shortest tree algorithm for optimal assignments across space and time in distributed processor system, *IEEE Transactions on Software Engineering*, SE - 7, 6, 1981.

[4] Bokhari, S. On the mapping problem. *IEEE Transactions on Computers*, C-30, 3, 1981, pp. 207 - 214.

[5] Casavant, T.L. and Kuhl, J.G. A Taxonomy of Scheduling in General Purpose Distributed Computing Systems, *IEEE Transactions on Software Engineering*, SE - 14, 2, 1988, pp. 141- 152.

[6] Chou, T. and Abraham, J. Load Balancing in Distributed Systems, *IEEE Transactions on Software Engineering*, SE - 8, 4, 1981.

[7] Chowdhury, S. Greedy Load Sharing Algorithm, *Journal of Parallel and Distributed Computing*, 9, 1, 1990, pp. 93 - 99.

[8] Chu, W. et al. Task allocation in distributed data processing, *IEEE Computer*, 1980, pp. 57 - 69.

[9] Chu, W. et al. Task allocation and precedence relations for distributed real-time systems, *IEEE Transactions on Computers*, 1987.

[10] Coffman, E. and Graham, R. Optimal Scheduling for Two-Processor systems, *Acta Informatica*, 1, 1972, pp. 200- 213.

[11] Eager, D.L., Lazowska, E.D. and Zahorjan, J. Dynamic Load Sharing in Homogeneous Distributed systems, *IEEE Transactions on Software Engineering*, SE - 12, 5, 1986, pp. 662- 675.

[12] Hua, K. Allocation of Processes and Files for Load balancing in Distributed systems, *Ph.D Dissertation, University of California at Berkeley*, 1985.

[13] Kim, S.J. and Browne, J.C. A General Approach to Mapping of Parallel Computations upon multiprocessor architectures, *Technical Report, University of Texas at Austin.*

[14] Lee, S.Y. and Agarwal, J.K. A Mapping strategy for parallel processing, *IEEE Transactions on Computers*, 1987, pp. 433 -

[15] Lo, V. Heuristic algorithms for task assignment in distributed systems, *Proceedings of the 4th International Conference in Distributed Computing Systems*, 1984, pp. 30 - 39.

[16] Ngai, T. et al. Mapping between Parallel Processor Structures and Programs, *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, 1987, pp. 172 - 181.

[17] Ramamoorthy, C.V. et al. Optimal Scheduling strategies in a multiprocessor system, *IEEE Transactions on Computers*, C- 21, 2, 1972, pp. 137- 146.

[18] Ramamritham, K. and Stankovic, J. Dynamic Task Scheduling in distributed hard real-time systems, *Proceedings of the 4th International Conference of Distributed Computing Systems*, 1981, pp. 96 - 107.

[19] Shen, C. and Tsai, W. A Graph matching approach to Optimal assignment in Distributed Computing Systems Using a Minima criterion, *IEEE Transactions on Computers*, C- 34, 3, 1985, pp. 197- 203.

[20] Shirazi, B., Wang, M. and Pathak, G. Analysis and Evaluation of Heuristic Methods for Static Task Scheduling, *Journal of Parallel and Distributed Computing, 10, 1990*, pp. 222 - 232.

[21] Tantawi, A.N. and Towsley, D. Optimal Static Load balancing in Distributed Computer Systems, *Journal of the ACM*, 32, 1985, pp. 445 - 465.

[22] Wang, Y. and Morris, R.J.T. Load Sharing in Distributed Systems, *IEEE Transactions on Computers*, C - 34, 3, 1985, pp. 204- 217.

☆U.S. GOVERNMENT PRINTING OFFICE: 1995-610-126-50122

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.